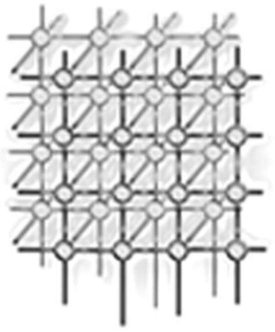
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2005; **17**:1271–1316
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.893



Architectural specification for massively parallel computers: an experience and measurement-based approach[‡]

Ron Brightwell[1,*,†], William Camp[1], Benjamin Cole[1], Erik DeBenedictis[1], Robert Leland[1], James Tomkins[1] and Arthur B. Maccabe[2]

[1]*Sandia National Laboratories, Scalable Computer Systems, P.O. Box 5800, Albuquerque, NM 87185-1110, U.S.A.*
[2]*Department of Computer Science, University of New Mexico, Albuquerque, NM 87131-0001, U.S.A.*

SUMMARY

In this paper, we describe the hardware and software architecture of the Red Storm system developed at Sandia National Laboratories. We discuss the evolution of this architecture and provide reasons for the different choices that have been made. We contrast our approach of leveraging high-volume, mass-market commodity processors to that taken for the Earth Simulator. We present a comparison of benchmarks and application performance that support our approach. We also project the performance of Red Storm and the Earth Simulator. This projection indicates that the Red Storm architecture is a much more cost-effective approach to massively parallel computing. Published in 2005 by John Wiley & Sons, Ltd.

KEY WORDS: massively parallel computing; supercomputing; commodity processors; vector processors; Amdahl; shared memory; distributed memory

1. INTRODUCTION

In the early 1980s the performance of commodity microprocessors reached a level that made it feasible to consider aggregating large numbers of them into a massively parallel processing (MPP) computer intended to compete in performance with traditional vector supercomputers based on moderate numbers of custom processors.

*Correspondence to: Ron Brightwell, Sandia National Laboratories, Scalable Computer Systems, P.O. Box 5800, Albuquerque, NM 87185-1110, U.S.A.
†E-mail: rbbrigh@sandia.gov
‡This article is a U.S. Government work and is in the public domain in the U.S.A.

Contract/grant sponsor: U.S. Department of Energy's National Nuclear Security Administration; contract/grant number: DE-AC04-94AL85000

Received 18 February 2003
Revised 6 November 2003
Accepted 10 November 2003

One basic argument for the commodity MPP approach was that its building blocks had very high performance relative to their price because they benefited from economies of scale driven by a consumer market that was huge in comparison to the niche market for specialized, high-end scientific supercomputers. Another argument was that it was becoming increasingly difficult to extract higher performance from a small number of vector units because this required higher clock rates and faster logic, and these in turn compounded the cooling problem. The solution, it was argued, was to distribute the computation spatially across a larger number of slower but cheaper and more easily cooled microprocessors.

The primary counter arguments were that it was conceptually difficult and economically infeasible to rewrite complex scientific and engineering applications to execute on MPP machines because they used a different computational paradigm, that of distributed memory and message passing. Furthermore, it was claimed, doing so would not bring the benefits sought because, in practice, customized vector processors could deliver a substantially higher fraction of peak performance, and that this more than compensated for the supposed price advantage of the commodity microprocessors.

Amidst vigorous debate on these issues, the commodity MPP research and development agenda was pursued aggressively at laboratories in the U.S. and Europe in the ensuing decade. By the mid-1990s the commodity MPP approach was very well established, and had become the dominant philosophical approach within the Department of Energy (DOE), one of the few leading consumers of supercomputing within the U.S. government and, indeed, the world. Hence, DOE's Accelerated Strategic Computing Initiative (ASCI) selected machines based on the commodity MPP model for its major procurements. Sandia National Laboratories' ASCI Red [1] was the first such machine, and the Red Storm system, currently under construction by Cray Inc., is the fifth and latest such machine.

It seemed that, in practice, the debate had been settled within the DOE and that, more broadly, a fairly stable détente had been reached: certain technical communities preferred and had more success with the traditional vector approach, in particular the climate modeling community and portions of the intelligence community. The traditional engineering community, which relied primarily on finite element and finite difference methods, had largely gravitated to the commodity MPP approach at the high end, and was becoming enamored of its offspring, the commodity-based Linux cluster, for mid-range computing.

The advent of the Earth Simulator [2] has dramatically reinvigorated this debate. Impressive results have been reported on at least three problems of long-standing scientific interest: direct numerical simulation of turbulence [3], modeling of Rayleigh–Taylor instability in plasma [4], and climate modeling via a spectral atmospheric general circulation model [5]. These applications ran at between 37 and 65% of the Earth Simulator's theoretical peak performance of 40 trillion floating-point operations per second (teraflops or Tflops), performance far in excess in both relative and absolute terms of any reported on the most capable commodity-based MPP machines in the world.

We should note immediately that, with over 5000 processors, a purpose-built network, a large distributed memory, and support for the message-passing paradigm, the Earth Simulator is surely an MPP machine. Hence, the essence of the debate centers around its use of low-market volume vector processors designed specifically for scientific computing rather than the more traditional choice for MPPs of high-market volume super-scalar processors designed for general use. In many respects, therefore, the Earth Simulator represents a convergence of these two basic approaches.

Our main purpose in this paper is to evaluate the effectiveness of this hybrid approach with respect to a timely, efficient, and economical solution of the engineering problems of most interest to Sandia.

We do so by comparing the performance of relevant application codes on a variant of the vector processor used in the Earth Simulator with that obtained on a commonly available microprocessor, and applying historical data and modeling techniques to infer bounds on the performance that would be observed when running these codes on the Earth Simulator and the Red Storm architecture. The application codes considered are used widely throughout the DOE and the Department of Defense, and are representative of codes in which the mainstream computational engineering community has a strong interest. Hence, we believe this comparison has wider significance.

The rest of this paper is organized as follows. In the following section, we discuss the evolution of the Red Storm architecture. Section 3 details the Red Storm hardware architecture, followed in Section 4 by a description of the software architecture. We provide a brief overview of the Earth Simulator in Section 5. We present a performance comparison of Red Storm and the Earth Simulator on benchmarks and select applications in Section 6. Section 7 compares the two platforms using a predictive performance model, and we conclude in Section 8 with an overview of the important contributions of this work.

2. EVOLUTION OF THE RED STORM ARCHITECTURE

In this section, we discuss the important system architecture issues and characteristics that influenced our decisions for Red Storm. We present our design philosophy and emphasize the importance of maintaining a balanced system architecture.

2.1. Overview of previous and current high-end architectures

Today's supercomputing choices are the product of commodity market competition, technology evolution, historical hardware and software legacies, and leadership choices within industry. Initially, all computing was high-end computing. Beginning in the 1950s with the IBM-7xxx series and Univac-11-xxx series of Von Neumann uniprocessors, up to the first minicomputers, building computers was essentially an expensive tour-de-force and their purchase could only be justified for the most compelling scientific and technical problems and commercial applications with the greatest payoff. This approach to supercomputing led, in the mid to late 1960s, to the pipelined architectures, typified by the CDC-6600, followed in the 1970s by the first vector system, the Cray-1, pioneered by Seymour Cray at Cray Research. This line of development was extended in the U.S. to include vector multiprocessors, first seen in the Cray X-MP and extended into the Cray Y-MP, the Cray C-90, the ETA-10, the Cray T-90, and the IBM 390 series. In Japan, the early models of the NEC SX series of parallel vector supercomputers (PVPs) exemplified the vector line of development.

The evolution of vector supercomputers led to the addition of special gather-scatter hardware for non-uniform memory references and to increasing numbers of processors within a coherent, shared-memory architecture. The most successful vector processors (e.g. the Cray X-MP, Y-MP, and C-90) were also excellent scalar processors—a fact that will be drawn out further below. These machines were largely used in a space- and time-sharing mode and—due to the low efficiency and cumbersome nature of their shared-memory parallel programming tools—only occasionally used for true multiprocessor parallel execution of jobs. Nevertheless, PVPs were the king of the performance hill until the early 1990s.

In the late 1970s, mini-supercomputers such as the VAX family and the Alliant family of machines began to provide systems with much lower cost (and lower peak performance) than the vector supercomputers. These machines shared many of the programming features and multi-user capabilities of the high-end PVPs. It was not unusual to have a VAX being used by 30 or more users simultaneously in a time-sharing mode.

In the early 1980s, personal computers and workstations evolved out of the minicomputer marketplace. Rapid advances in CMOS integrated circuits led to the appearance of computers on a chip, as in the MicroVAX and its Data General competitors. The minicomputers became personal technical computers with the advent of Sun Microsystems and Apollo workstations. It was not long thereafter that the first personal computer (PC) aimed at the business and technical marketplaces arrived. Personal computers powered by commodity processors from Motorola (68xxx family) and Intel (80xx family) began to be used for technical endeavors and early on began to compete with the Sun and Apollo offerings, since their huge cost advantages allowed them to be deployed much more broadly. By the late 1980s, technical America largely ran on Intel-based IBM PCs (although IBM had long ceased to be the sole source for these machines) or Motorola-based Apple Macintoshes.

PVPs were then dealt a massive blow from which they never truly recovered. First, the same processor, memory, and board technologies that powered PCs were used to create massively parallel computers with dozens (Intel iPSC with up to 64 processors), to hundreds (nCUBE-10 with a maximum of 1024 processors), to thousands (CM-200, CM-5, nCUBE-2, Intel Delta and Intel Paragon) of commodity or commodity-like processors. These machines were difficult to program at first, but they were shown quickly to be capable of consistently high performance across a wide variety of applications. They were also much cheaper than the PVPs. By the second generation of MPPs, typified by the Cray T3E and ASCI Red, PVPs were no longer leading edge. They did, however, show up as a node choice in Japanese MPPs, for example in the NEC SX-4 and in the Fujitsu VPP-500. Cray Inc. has resurrected PVPs as the node of their current-generation X-1 MPP, and NEC continues to use them in the SX-6 and Earth Simulator MPPs.

The second large blow to PVPs came from the development of semi-commodity servers based on workstation technology: Sun Microsystems, Digital Equipment Corporation (DEC), Hewlett-Packard (HP), IBM, and Silicon Graphics, Inc. (SGI) all introduced shared-memory servers based on their various workstation lines. These processors provided a similar programming and execution environment to PVPs (porting codes was generally not difficult), and they were within reach of many organizations. Because of their ability to leverage more commodity technologies, they were much more cost-effective in many cases than PVPs. In addition—and this was critical in attracting Independent Software Vendors—codes worked identically on both the workstations and the servers, producing bit-for-bit the same results without a recompile. By taking advantage of new super-scalar instruction sets and large caches, these servers were able to overcome some of the architectural balance advantages of the PVPs.

The next step in high-end computing was the introduction of cluster technologies. This has proceeded on two fronts: in the first development, beginning in the late 1990s, IBM, Compaq, and SGI, among others, began creating proprietary clusters using their shared-memory servers and custom-designed or semi-commodity networks. The 3+ Tflops ASCI Blue Pacific computer system built by IBM for Lawrence Livermore National Laboratory (LLNL) and the 3+ Tflops ASCI Blue Mountain system built by SGI for Los Alamos National Laboratory (LANL) were the two largest such clusters built in the late 1990s. They were, respectively, the second and third terascale systems ever built—the first having been the 1.8 Tflops ASCI Red system, which was an MPP built by Intel for Sandia. ASCI Red was subsequently upgraded to 3+ Tflops as well. Since then, the 12 Tflops IBM ASCI White machine and the 30 Tflops Compaq ASCI Q machine have been acquired by LLNL and LANL, respectively. The IBM approach has been to use their own custom multi-level switch fabrics to interconnect shared-memory nodes based on their Power workstation processors. These nodes had four processors in the case of Blue Pacific and 16 processors in the case of White. SGI utilized its MIPS R10000 processor in non-uniform memory access, 128-processor shared-memory nodes. They connected their nodes with HIPPI-800 switch fabrics. (The planned upgrade to HIPPI-6400 was not successful.) For the ASCI Q machine, Compaq utilizes its ES-45 four-processor shared-memory servers based on the Alpha EV68 processor. Both IBM and Compaq sold similar terascale systems to a variety of other customers. Oak Ridge National Laboratory, the Swiss National Computing Center at Manno, and Lawrence Berkeley National Laboratory purchased IBM systems. Pittsburgh Supercomputing Center and the Centre de l'Energie Atomique in France purchased Compaq systems.

At the same time, in the mid to late 1990s and continuing to the present, true commodity clusters were being built and deployed based on uniprocessor or dual-processor nodes utilizing Compaq Alpha or Intel x86 processors. These clusters used mainly semi-commodity Myrinet [6] interconnects from Myricom; but smaller examples sometimes were based on gigabit (or slower) Ethernet switch fabrics. In all cases, the system software was built around the Linux open-source operating system. The first large clusters, for example the Computational Plant (Cplant^{TM}) [7] cluster at Sandia and the Chiba City [8] cluster at Argonne National Laboratory, developed their own runtime environments and input/output (I/O) file systems. These systems proved extremely effective at providing supercomputing at the best price-performance characteristics yet achieved. At Sandia, at least, the programming environment and runtime environment were very similar to that provided on MPP machines [9]. The cost of operation and availability was also similar to experience with MPPs. The first terascale commodity cluster was the Alpha processor-based 2500-processor Cplant^{TM} Antarctica cluster at Sandia. Antarctica, which was deployed in 2000, ran Linux with Sandia's Cplant^{TM} environment. Currently, the total Cplant^{TM} cluster capacity at Sandia exceeds 3 Tflops. Subsequently, a number of institutions have installed even larger clusters, including Intel x86-based clusters at LANL and LLNL. In one notable development, the Lawrence Livermore cluster utilized a Quadrics [10] fat-tree switch fabric rather than the Myrinet fabrics utilized at Sandia and LANL.

As we shall discuss below, none of these clusters—custom or commodity—have system balance between computation and communications that is competitive with that found on true MPPs such as the Cray T3E and the Intel ASCI Red. Nonetheless, for several important classes of applications, they are capable of achieving reasonably high parallel efficiency on a thousand processors or more. They also, in general, lack full-system reliability, availability and serviceability (RAS) features. For truly large systems, this has caused difficulties in running large jobs with long execution times. In addition, of the large clusters deployed until recently, only Cplant^{TM} has truly scalable system software [9]. Cplant^{TM}, like ASCI Red and the Cray T3E, is designed to provide service to dozens of simultaneous users and has fast, scalable system boot-up, and executable loading capabilities. By contrast, all of the commercial clusters described above have slow, non-scalable boot-up mechanisms (on some commercial clusters, it can exceed 10 h to do a full boot of the system; by contrast the system boot on ASCI Red is less than 2 min and for Cplant^{TM} it is less than 15 min).

Also today, a number of new-generation systems are being developed. They include ASCI Purple, a next-generation shared-memory cluster from IBM with 64 Power-4 processors (each 8 Gflops) per node and a new Federation switch fabric. This machine will be deployed in 2005 at LLNL. Its peak speed will be at least 60 Tflops. In addition, IBM is developing a new MPP, Blue Gene/L, with several thousands of low-power processors connected by a mesh switch fabric. Although this machine is not highly balanced, it will have a very high peak performance and could achieve significant performance on certain limited classes of scientific applications.

HP, which recently acquired Compaq, is producing a new semi-commodity cluster based on Intel's 64-bit Itanium II processor. It utilizes two-way shared-memory nodes, and its interconnect is a Quadrics fat-tree. HP continues to sell the Alpha ES-45-based clusters formerly developed by Compaq. However, they have not announced plans to build clusters based on the Alpha EV-7 processor, the Compaq-developed replacement for the EV-68, and one of the best-balanced processors developed since the early Cray vector processors. Similarly SGI is offering an Itanium II-based cluster that is distinguished by a very large non-uniform memory access shared memory node with 64 processors per node. Both the HP and SGI systems will run Linux as their operating system.

Finally, Cray Inc. has developed and begun selling a new MPP with PVP nodes, the Cray X-1. The X-1 has four, four-way vector processors per node. Each vector processor has two vector pipes, each capable of two operations per clock cycle, which share a data cache and a common scalar unit. The vector processor clock speed is 800 MHz and the scalar clock is 400 MHz. The network topology is a complex modification of a three-dimensional toroidal mesh [11].

2.2. Red Storm design philosophy

Our design philosophy is captured in the acronym *SURE*. SURE stands for scalability, usability, reliability and economy. Since processors, I/O systems and memory systems cannot be made arbitrarily powerful, we have no choice but to employ massive parallelism in creating ever faster compute engines. For that approach to be effective, all aspects of the system—processor, memory and interconnect hardware, I/O hardware, power and cooling, system volume, and all aspects of system software (including system boot and job loaders)—must be designed for *scalability*. For performance, scalability means that system software overheads are low and that system performance increases linearly—and nearly perfectly—as we increase the number of processors up to the full system size. System boot and application load times must increase no more than logarithmically with the number of processors and must be measured in seconds or minutes not hours. In addition, system I/O performance must not depend strongly on how many processors are requesting I/O services. Of course, these system characteristics must, at a minimum, not prevent the development and use of scalable applications software. Scalability, in general, requires that we examine the design with respect to every aspect of system operation and ensure that bottlenecks are removed. For Red Storm, one scaling requirement of the design was that major applications achieve over 50% parallel efficiency using all 10 000+ compute nodes in the system.

The system must be *usable* for the intended end-use. This means that it provides a simple, effective runtime environment, compilers and libraries for the major languages, and support for major, community-standard parallelization tools (today that essentially means MPI [12] and MPI-2 [13]), effective I/O capabilities and file systems. The runtime environment should provide scalable multi-user support (we need to serve dozens of simultaneous users) and the ability to provide the aspects of a single-system image needed for applications programmers to manage their jobs and for system managers to effectively manage the entire resource. It means that job allocation mechanisms need to be flexible and efficient and that the hardware design must not prevent that flexibility and efficiency. It means providing effective debugging capabilities and performance measurement and optimization tools. Users and system managers need to see capabilities equivalent to a full UNIX environment for their interactions with the system. However, it does not mean adding things that are not useful to running high-performance parallel applications. This approach drives us toward a partitioned system software environment. The ability to do word processing or text editing, send email, open sockets, or migrate processes does not add value to or belong on the compute nodes of such a system. (See further details in Section 4 below.)

The system must be *reliable*. We desire for Red Storm that a typical large job be able to run, on average, for 100 h without failure of the job. For a small system, reliability in hardware can be achieved by utilizing good design approaches and high-quality manufacturing practices. In the case of the huge systems Red Storm is aimed at, reliability must be an explicit design feature of the system. Roughly speaking, for independent random failures, the reliability of a system is inversely proportional to the number of replicated parts. So, if the mean time between failure of a single power supply is N h, the mean time between failure of at least one power supply out of 10 000 is to high accuracy, $N/10\,000$. It is important to define what we mean by reliable. We do not mean high availability. Availability is crucial to many commercial and system-critical computing functions—to the point that complete redundancy is often utilized to preserve guaranteed availability. It is not crucial to our form of strategic computing. Reliability in this context means that a large parallel job running for many hours has a high probability of successful completing. It is measured by the mean time between job failures. Note that the system can undergo a failure that does not lead to loss of a job without affecting reliability—this is important to developing reliability enhancement strategies. A related requirement would be that if the system undergoes a failure that is local, only jobs using that local resource are affected. This kind of aspect of reliability we also call resiliency. Note that a system can have very high availability and not be reliable for our purposes. It is, by contrast, unlikely that a system that has low availability could have high reliability. Finally, for a hardware system to be reliable, it must be serviceable: a system that has infrequent failures, which take many hours to identify and fix, is not reliable.

Note that, in a system designed for hardware reliability, software reliability becomes the limiting factor in achieving high reliability. Software reliability is at best an inexact science. Our experience has shown that systems with nearly identical software features can differ wildly in reliability. (For obvious reasons, we do not cite examples.) Software reliability is first and foremost inversely proportional to some power of the number of lines of code. It is also inversely proportional to the inherent complexity of that code. It is greatly enhanced by rigorous design and engineering processes. So our mantra in achieving high software reliability is ‘simplicity in design first, followed closely by rigor in development’. To this end we strongly prefer partitioned software environments (not just for reliability, also for performance and scalability). We prefer to utilize complex, full-featured system software only where it is absolutely needed (mainly for log-in, system services and I/O) and to utilize extremely simple but powerful (for that purpose) ultra-lightweight software on the great majority of the nodes in the system (the compute partitions).

Finally, we seek a system that is *economical* to purchase and own. This affordability aspect includes cost of purchase, cost of maintenance, cost of operations and cost to the user of using the system. The single best thing we can do to control expense is to keep the system simple. It lowers manufacturing cost; it increases inherent reliability; and it makes it less expensive to maintain and operate. Another key is to utilize high-volume commodity parts wherever feasible. This not only reduces cost of parts, it allows use of standard manufacturing processes which are less expensive (and more reliable) than custom processes. We cannot simply use commodity technologies everywhere: commodity technologies are not designed with scalability and high reliability at scale as design goals. While redundancy and other reliability features can drive up initial costs—and if not properly managed can cause complexity-driven failures—they can increase greatly operational efficiency and flexibility as well as reducing the cost to the user community in terms of lost work from unnecessary failures. Systems designed to operate at many Tflops often require megawatts of power and cooling. So, power and cooling constitute major components of the cost of ownership. First, there is the obvious cost of purchasing electricity, which for systems on the scale of Red Storm can add up to millions of dollars per year. However, if power requirements exceed installed delivery capabilities, the capital cost of bringing in more power can be significant (millions of dollars). Similarly, cooling adds to the power budget. Again, however, it also adds to the capital facility costs. More chillers and heat exchangers may be required. They have their own cost, but they may also require larger volumes that may lead to millions of dollars in facility expansion costs. Finally, if the facility costs to accommodate added cooling become too large, or if air-cooling becomes thermodynamically infeasible, it may be necessary to go to liquid-cooling methods in the computer itself. This could have enormous impact on cost of purchase and complexity of operations and maintenance. Therefore, it is critical to minimize power requirements. In Section 3, we describe how we attempted to minimize the power and cooling budgets for Red Storm. System volume is another major component of cost. For example, the Earth Simulator is so large that it requires a truly huge, multi-story facility to house it [2]. Similarly, the ASCI White and ASCI Q machines at LLNL and LANL, respectively, required the building of large new facilities to house them. In the case of Red Storm, the space budget was very limited (under 1000 m^2). This meant that we were prevented from using any currently offered commercial cluster solution on the basis of density alone.

In the remaining parts of this section, we will discuss architectural tradeoffs, including balance in the processor and memory sub-systems, balance between processor speed and interconnect, topological design choices, software issues, and RAS.

2.3. Node and processor choices

As mentioned above, in designing a new system, we are faced with a number of choices for the nodes within the architecture (assuming that we stick with existing building blocks rather than moving to a radically new design in which the concept of individual nodes becomes blurred). We could have a uniprocessor node or a multiprocessor node. We could choose a vector processor, a semi-commodity super-scalar workstation processor, or a commodity superscalar processor. Issues that become important in the choice involve cost, cost-performance, balance, scalability, and programming model.

2.3.1. Processors per node

There are two competing issues in this decision area: on the one hand, memory is a major cost factor in large machines. In an MPP, global data must be replicated on each node of the system. Otherwise, processes running on a given node can be heavily dependent on distant data. Relying unnecessarily on distant memory accesses leads to excessive overhead, load imbalances and loss of scalability. On the other hand, if a shared-memory multiprocessor node becomes too large, there is a danger that the hardware cost of the node will rise unacceptably due to the N^2 cost of cache coherency and fast memory accesses. In any case, performance can suffer due to computational cost of cache coherency and the overhead due to memory contention in a shared-memory programming model as well as contention for operating system services. Finally, excessively large shared memory nodes either make the cost of the system interconnect rise or lead to system imbalance.

The issue of memory waste through duplicated data in uniprocessor-based MPPs is not as much of a problem as it once was. When typical single-processor memories were in the megabyte to tens of megabyte ranges, the cost of duplicating the operating system for every processor was unacceptable. In fact, this was one of the forces leading to the development of ultra-lightweight operating system kernels (LWKs) for the compute processors in an MPP [14,15]. If global databases had to be replicated (for example, material properties for a radiation hydrodynamics calculation in simulations of stellar evolution), the problem only worsened. Today, when a typical single-processor memory bank ranges in size from 500 MB to 8 or more GB, this has become much less of an issue. (Although the computational overhead drivers for LWKs have not gone away.) Replicated data now probably represent no more than 10% of the memory on a uniprocessor MPP for typical scientific applications.

The issue of computational overhead in shared-memory multiprocessors is also not as much of a problem as it once was. There now exist optimistic cache coherency mechanisms that lessen the computational cost of maintaining coherent memory state, at least for modest numbers of processors. The overhead due to memory accesses in a shared-memory programming model remains an issue. Basically, all modern shared-memory multiprocessor systems are based on a non-uniform memory access (NUMA) model. In this model, memory access overhead and bandwidth are not independent of location in the node. Nonetheless, shared memory models treat them as though they were. (And, even if they were 'flat' memory spaces, memory access contention grows in proportion to the number of processors.) These effects lead to loss of efficiency that can become severe as the size of the node increases.

To some extent, modern multi-threading capabilities, e.g. OpenMP [16], allow the programmer to code to optimize for locality of reference. Nevertheless, there remain two issues with that approach. First, OpenMP requires a global address space not provided by most MPPs. Second, it does not scale to hundreds or thousands of processors. In practice, this means that a programmer wanting to use shared-memory directives or a threads-based language is faced with the dilemma of having to use two programming methodologies: shared memory coding for processes within a node, coupled with MPI or another message-passing protocol for inter-node processes. In some sense, this is like having the worst of both worlds. For that reason, among others, most shared-memory-based MPPs provide an MPI mechanism for memory accesses within a node as well as between nodes. A third issue with shared-memory programming models is the phenomenon of code explosion: in the case of directives-based shared memory methods, the amount of code in directives and the complexity of those directives add tremendously to the difficulty of code development and debugging. They are also not natural tools for maintaining locality of memory references. This of course makes it harder to achieve high efficiency. By contrast, distributed-memory message-passing codes (and related languages) do not add appreciably to the serial code needed to run on any processor. While that is not to say that achieving high performance in MPI codes is now routine, it does mean that most of a program's code base can remain untouched. For shared-memory systems based on semi-commodity or commodity processor and memory technologies, the memory bandwidth per processor often decreases as the processor count increases. This bandwidth sharing will have detrimental effects on performance for bandwidth-sensitive applications.

Two final issues face shared-memory multiprocessor-based MPPs. First, operating system services are hard to distribute evenly and effectively. For that reason, a processor is often essentially dedicated to OS services. Competing for services from that processor can become the bottleneck in large shared-memory multiprocessor nodes. Another issue is balance. A large multiprocessor node has, by its nature, a very large computing capability. That means it requires a high-bandwidth interface into the system communications fabric and that the communications fabric needs to have correspondingly high bandwidth. There are three simple ways to do this: a multi-rail system as is used, for example, in some Quadrics-based systems, or a large router switch with multiple ports per node is used; or a very powerful network interface and an equally powerful router are employed. The first two solutions have not yet been applied effectively in any truly balanced system. The full crossbar switch in the NEC Earth Simulator typifies the third solution. Such a solution is costly since it stresses technological capabilities and is not able to leverage commodity or semi-commodity industrial technologies. All three solutions have associated software scheduling issues: in the first and second solutions, unless a port per processor in the node is provided (this is usually impractical for large shared-memory machines) the network requests of various processors have to be scheduled effectively over the several ports of the interconnect. In the third solution, the requests of all the processors have to be arbitrated by the network interface.

This analysis, coupled with our positive experience with uniprocessor and small (two-processor) nodes, leads us to favor a uniprocessor design, although a two- or four-processor node would possibly be just as effective.

2.3.2. Vector, semi-commodity, or commodity processors

The issue of processor choice is somewhat dependent on the application space in which the system will be used. For example, vector processors have long had superior performance on weather and some climate codes. The gather-scatter capabilities of classic vector architectures like the early Cray machines had real advantages for pattern-matching applications such as those arising in cryptology. Commodity processors have large, often multi-level, reasonably fast caches; and so, codes optimized for data locality and cache re-use do extremely well on them—especially when cost per unit computing is taken into account. Several semi-commodity processors are also available. Interestingly, where vendors offer commodity and semi-commodity processor families, the semi-commodity processors tend to be less cost-effective on many codes than their commodity siblings. We give an example of this below in the case of Intel's IA-32 commodity line and IA-64 semi-commodity processors. For all processors, the performance on technical applications is typically more dependent on the memory hierarchy (bandwidth, latency—including page-miss latency—and the number of registers) and the efficiency of instruction execution, and less dependent on the speed of the arithmetic and logical functional units.

All modern processors rely on a great deal of fine-grained parallelism. Nearly all are pipelined, and many utilize look-ahead and speculative execution. Most are, in principle, capable of carrying out multiple instructions per clock cycle and have multiple execution units to support that. All current vector processors have an associated scalar unit to carry out non-vector operations. Today's vector machines are also multi-pipe machines. That is, they rely on being able to move data through several vector execution units in parallel. Generally, these units are programmed in a lockstep or single-instruction-multiple-data (SIMD) method. The advantage of multiple pipes is that the vector units are able to get higher peak performance at lower processor speeds. Note, however, that the requirements on memory latency are actually made more stringent with multiple pipes, since instead of paying the latency penalty for starting up a vector for one pipe, it must be paid for all the pipes. In addition, the memory bandwidth requirements are not reduced at all by multiple pipes. Also note that the overall performance of a vector architecture for many applications is as dependent on having a fast scalar processor as it is on having fast vector units. This is because for many applications—generally the more realistic ones—the average vector length can be quite short. Real problems tend to encounter branching conditions quite often. In engineering mechanics, for example, real problems tend to have irregular domains and many different materials leading to numerous conditionals and poorly vectorizing codes. It is true that nearly all of the first- and second-generation codes in science and engineering (so-called legacy codes) were developed for vector architectures and that they often achieved as little as 15–25% of the theoretical peak on those architectures for realistic problems. It is also true that the fact that early Cray vector architectures had scalar computing capabilities that were also fast (in comparison to their vector units) and low latency and high bandwidth to memory enabled those relatively high efficiencies. Unfortunately, most of today's vector architectures have relatively slow scalar units. Finally, nearly all the legacy codes have by now been re-optimized for cache-based commodity and semi-commodity processors; and many would have to be re-architected to do well on vector processors. This is less likely to occur for most commercially supported codes from ISVs than it is for academic or research codes.

To understand the importance of scalar units, consider the following simplified application of Amdahl's Law [17]. Let us compare a vector processor to a (super)scalar processor. Assume that the vector processor is N times as fast as the scalar processor on vector work and is $1/M$ times as fast on scalar work. Let the scalar speed be s. Finally, assume that a fraction, p, of the total work to be done can be vectorized and that the remaining fraction, q $(= 1 - p)$, must be done in scalar mode. Then the vector speedup (or slowdown), S, is defined by the time it takes for the scalar processor to carry out the task divided by the time it takes the vector unit to do so.

$$\begin{aligned} S &= T_{\mathrm{S}}/T_{\mathrm{V}} \\ S^{-1} &= [pW/(sN) + (1-p)W/(s/M)]/[W/s] \\ S &= 1/[p/N + M(1-p)] \end{aligned}$$

If we consider a Pentium-4 commodity processor running at about 2 GHz and the Earth Simulator processor, we could reasonably assume that $M = N = 4$. This is because the peak speed of the Earth Simulator vector unit is 8 Gflops compared with a peak of 2 Gflops for the Pentium-4. At the same time, the scalar speed of the Earth Simulator is about 500 Mflops. (This is an over-simplification but not an unreasonable one.) If that holds, we can assume that

$$S = 1/[p/4 + 4(1-p)]$$

This means that for $p < 0.8$, the Pentium-4 will actually be faster than the Earth Simulator on that problem.

Now this analysis is clearly oversimplified: we have assumed single, typical vector and scalar performance for the Earth Simulator, and a single performance number for the Pentium-4. In reality, the vector speed achieved on the Earth Simulator will depend on operations mix and vector length. Here we have basically assumed long vectors. Similarly, the performance of the Pentium-4 will depend on operations mix and how well the data fit into and reuse cache. Nonetheless, the numbers used are not in contradiction to our actual experience (see Section 6). Note that this model implies that the vector processor on the Earth Simulator is 16 times as fast as its scalar unit, which reinforces the point about lack of balance between vector and scalar capability in modern vector systems. This model also indicates that if 80% of the work were done in the vector unit, 80% of the time on the Earth Simulator for such a problem would be spent in non-vectorizable (scalar) operations. Again, this is consistent with our experience. Unfortunately, this shows how difficult it is to achieve high vector performance: essentially all the operations have to be vectorized or vectorizable and the vector lengths need to be long. At Sandia, codes tend to be used on complex problems with complicated geometries and many materials, leading to irregular local memory references and many conditionals. This tends to make vector architectures less favorable for a broad class of our applications. By contrast, a typical code modeling fluid flow in the atmosphere would involve simpler material properties and a very regular domain; and hence most of its work would be in long vectors. For such problems, the Earth Simulator could have a marked performance advantage.

Below we shall see that, in contrast to inner-loop (vector) parallelism, for outer-loop distributed-memory parallelism, Amdahl's Law is much less restrictive of performance. This is because most scientific problems have a large degree of data parallelism and are intrinsically local. The fine-scale irregular memory references that hurt inner-loop parallelization are replaced by infrequent, more regular, communications patterns on the larger scales important to outer loop parallelism.

It is important to examine both performance on relevant applications and the cost of that performance. In general, vector processor technologies have tended to be an order of magnitude more expensive than are commodity processor technologies in absolute terms. Unless that cost disadvantage is compensated by a compelling performance advantage, vector systems will be uncompetitive. We shall discuss detailed performance comparisons between processor families in Section 5. For our current purposes, it is useful to augment our benchmarking and to summarize some of its findings. A favorite benchmark for high-performance technical computing is the SPECfp2000 set of benchmarks from the SPEC CPU2000 suite [18]. The SPECfp2000 suite contains a variety of applications. One application is particularly compelling in predicting the performance of processors on real science and engineering mechanics calculations. The fma3d benchmark is meant to be typical of the expected behavior of important finite-element mechanics codes. We show the rounded results of various commodity and semi-commodity processors on fma3d in Table I.

In Table I, we show both the absolute fma3d ratios and the normalized ratios in which we have divided the fma3d ratios by the peak speeds of the processors. It should be noted that we have only counted 1 flop per clock cycle for the two IA-32 processors, the AMD Athlon and the Intel Pentium-4. It is no surprise that the Alpha EV7 tops both the absolute and the relative performance list, since the EV7 has by far the highest memory bandwidth per flop and the lowest page miss latency, measured in clock cycles, of any of the processors shown. Perhaps more surprising is that the two least expensive processors, the commodity Athlon and Pentium-4, have competitive absolute performance and that their relative performance is only bettered by the three Alpha processors, the EV6, EV68 and EV7. On a per-dollar basis, these two processors have by far the best performance. Equally surprising to some will be the fact that they both outperform the Itanium II in both absolute and relative performance on this benchmark, even though their costs are about an order of magnitude less. If we discount two SPECfp

Table I. Performance of various processors on fma3D from SPECfp2000.

Processor	Peak speed (Gflops)	fma3d ratio	Normalized fma3d ratio
IBM Power3	1.5	300	200
HP Alpha EV6	1.0	420	420
Intel Itanium II	4.0	776	190
AMD Athlon	2.25*	866	380
Intel Pentium-4	3.06*	1038	340
HP Alpha EV68	2.5	1120	450
IBM Power4	5.2	1020	200
HP Alpha EV7	2.3	1380	600

Note that the numbers are approximate.
For the AMD Athlon and the Pentium-4 we have assumed 1 flop per clock cycle.

benchmarks that are extremely cache friendly, the results for the rest of the SPECfp2000 benchmarks are similar.

In Section 6, we show that these results foreshadow the outcome of benchmarking these processor architectures on real science and engineering applications. The benchmarks in Section 6 also show that on a price-performance basis, commodity processors in general outperform the Earth Simulator vector processor on typical Sandia applications. In some cases, for example in the case of the EV7, the absolute performance is greater than that of the Earth Simulator processor.

It is worth outlining our experience on what determines processor performance based on over 30 years of dealing with processor efficiency issues. Key attributes include the following.

- Key enablers of high performance on real applications:
  - functional units:
    - good integer performance—HPC codes have many integer operations;
    - simple architectures—easier to compile for high performance;
    - short pipelines—deal with branches more effectively;
  - memory subsystems:
    - many registers—needed to take advantage of instruction parallelism;
    - simple cache hierarchies—multi-level caches increase latency;
    - support for large memory pages—avoiding TLB misses greatly reduces latency;
    - short cache lines—reduces latency;
    - wide paths and fast clocks into memory;
    - integrated memory controllers—good for both bandwidth and latency.
- What does not help real performance:
  - excessive instruction level parallelism—hard to utilize effectively;
  - multiple multiply–add units—good for LINPACK benchmark [19] and not much else;
  - compiler optimizations that often fail to compile or give the wrong results;

- failure to meet ANSI standards;
- complex architectures—these make compilation difficult and inefficient;
- specialized functional units that are difficult to schedule and feed with data.

The results in Table I can be rationalized on the basis of these observations. Such a discussion requires a report in its own right; so we forego a more detailed discussion here, and simply offer the above observations as a framework useful for analyzing the performance characteristics of a given application on a given processor.

In any case, as discussed in Section 3, these and similar considerations led us to rank the processor choices for Red Storm based on effectiveness for our design, ignoring cost for the moment, as follows:

1. HP Alpha EV7;
2. AMD Opteron;
3. Intel Pentium-4 and AMD Athlon;
4. IBM Power-4;
5. Intel Itanium II.

When we take into account performance relative to price, this ranking changes somewhat:

1. AMD Opteron;
2. Intel Pentium-4 and AMD Athlon;
3. HP Alpha EV7;
4. IBM Power-4;
5. Intel Itanium II.

Note that we could not include the Cray X-1 and the Earth Simulator in these results. This is because neither Cray nor NEC has published (as of February 2003) specFP2000 results for their processors. However, we do compare the performance of selected real applications on the closely related NEC SX-6 to that on semi-commodity and commodity processors in Section 5.

It should also be noted that we have not provided performance data for the AMD Opteron. We have carried out extensive testing of beta Opteron hardware. Those tests are under non-disclosure agreements and cannot be discussed in this paper. However, the complete architecture has been published by AMD. In addition, as of February 2003, AMD has released a preliminary overall SPECfp2000 ratio for the Opteron. According to AMD, it achieved a ratio of 1170.

The highest comparable number listed by SPEC as of February 2003 is for the Itanium II with a ratio of 1431 [20]. The large Itanium II number is largely due to outstanding performance on `swim` (3373) and `art`(4178). The `art` benchmark is a graphics benchmark that appears to fit entirely in cache; and `swim` is a cache-friendly, shallow-water fluid dynamics application. The SPECfp2000 results for the Itanium II are quite ‘peaky’ with a high ratio of 4178 on `art` and a low ratio of 726 on `mesa`. In contrast, the results for the two IA-32 architectures are much less peaky: the AMD Athlon shows a high of a little over 1200 on `swim`, `galgel` and `mesa`, and a low of a little under 600 on `art` and `ammp`; the Intel Pentium-4 (as reported by Dell) shows a high of 1800 on `swim` and a low of a little under 600 on `sixtrack`. The Opteron individual benchmark results, which have not been published, should show behavior close to that of the Athlon, based on architectural heritage and our experience in testing them on our applications.

Using performance weighted by cost as a metric, the Opteron was the clear winner in our architectural considerations. Also of importance to our considerations was its full backward compatibility with the IA-32 architecture. Our Pentium-4 IA-32 codes run without recompile on the Opteron. To this point in our testing, they run more efficiently. The architectural simplicity makes it quite likely that the full 64-bit compilers will compile quickly and produce quite efficient code. The fast, open HyperTransport [21] interface integrated into the Opteron was also an important consideration because it made the job of designing a fast, relatively inexpensive, low-latency interconnect for Red Storm much more feasible. Finally, the integrated memory controller reduces memory latency and cuts the cost and complexity of the chipset considerably.

2.4. Communication and topology

2.4.1. Communication mechanisms

Processor issues not withstanding, the fundamental reason why some architectures are less effective for large problems is due to the lack of balance between computational speed and the speed at which data can be delivered to and from the processor that needs it. This obviously includes local memory bandwidth and latency and, for multiprocessor nodes, cache coherency and memory contention issues. However, having dealt with these issues above, we now concentrate on distant memory accesses. We state, without further elaboration, since it is the experience of many in the community, that truly coherent, shared memory approaches to distant memory accesses do not scale using existing technological capabilities. So, at a minimum, the system must deal with distant memory accesses differently than it does with local memory loads and stores. There are currently two common methods for doing this. First, by using global address spaces and some test and set mechanism (e.g. lightweight barriers), distant memory can be accessed directly by its address. One example of this is the global address space technology underlying the SHMEM library on the Cray T3E [22]. Unless this method allows entire vectors to be loaded or stored based on a single instruction execution, it is obviously non-scalable. It should also allow for gather-scatter capabilities to deal with non-uniform-stride vectors. The Cray T3E provided both of these capabilities. The second method is by using an explicit message-passing mechanism, such as MPI.

In principle, the first mechanism can be enabled in software through message passing. In fact, doing so leads to unacceptable overheads, and so is not utilized in practice. So far, direct memory access mechanisms that have been applied have allowed an advantage of about a factor of 5–10 for single-word and short vector puts and gets relative to MPI message passing on the same architecture. There are national security applications for which direct-memory addressing capability, together with gather-scatter capabilities, is a real advantage. For most science and engineering applications, however, experience has shown that message lengths can be sufficiently large to reduce such an advantage to a minimal effect. In addition, nearly all modern, massively parallel applications are written in MPI, so MPI was a Red Storm requirement in any case. In our design studies, we made MPI and portions of MPI-2 a requirement, and noted that direct memory addressing and gather-scatter capability would be good future architectural improvements.

2.5. Balance requirements and network topology

For applications to be able to take advantage of MPP architectures, they must minimize the serial work fraction [17]; they must minimize the load imbalance; and they must also minimize the parallelization overhead. Overhead typically is dominated by synchronization and communications. For the architectures we are considering, synchronization occurs through explicit message passing, so we lump them together henceforth. Early on, Amdahl's Law was taken to imply that parallelization beyond a hundred processors was extremely difficult to achieve. In fact, on balanced MPPs, applications routinely achieve high parallel efficiency on thousands of processors when the job size is scaled up linearly with the number of processors. They achieve high efficiency on hundreds of processors for large fixed-size jobs. This is because the serial work can be controlled to be in the noise on most scientific and engineering applications. In addition, sophisticated load-balancing methods have been developed and implemented that achieve a high degree of load balance while simultaneously minimizing communications overhead. We expect to achieve typical parallel efficiencies of 70% or higher for our most important applications at the full system scale (10 000 or more processors). That we can do this is due to the inherent parallelism of most physically based problems: for the most part, nature is inherently local and inherently parallel. Nonetheless, we shall fail to achieve acceptable performance unless we design the communications capabilities of our system to match the processing and local memory speeds.

What is required for this balance depends on the network topology and on the scale of the system. Balance is achieved at the lowest required bandwidth relative to processing speed for a network that is a complete graph; that is, for a network in which every node has a bi-directional link connecting it to every other node. Clearly such a network has an $O(N^2)$ or greater cost, where N is the number of nodes. The most achievable way of providing such connectivity is a non-blocking crossbar switch. Other switch-based networks include multi-level, crossbar-based networks such as those provided by IBM in ASCI Blue Pacific and ASCI White machines. They also include fat-tree networks such as those previously used in the CM-5 from Thinking Machines Corporation and those provided by Quadrics (and its predecessor, Meiko) as well as similar Clos networks provided by Myricom. These high-connectivity trees have the property that the cross-section bandwidth is high and is preserved at all levels of the tree. However, they do not share the contention-free properties of the full crossbar. Fat-tree networks reduce this contention by the randomness of path selection. They have the advantage that the diameter of the network grows only as $\log(N)$ for N nodes. Nonetheless, they require $O(N \log N)$ network switches to connect N nodes. For large networks, this can be a marked disadvantage compared with mesh networks which require $O(N)$ switches. In addition, along with all other large-switch networks, they suffer from long wires and from wire-length induced latencies. They also have less locality than meshes, which can be a disadvantage for algorithms that take advantage of locality (most algorithms do). Butterfly networks share many properties with fat-trees. At least one major system was butterfly-based: that provided in the late 1980s and early 1990s by BBN Corporation. We do not discuss butterflies further because, where they differ, the features of fat-trees are superior.

Other low-diameter networks include hypercubes. These have a uniform $\log(N)$ diameter for N nodes. However, they require $\log(N)$ ports per processor which can become impractical for large systems. Also, for large systems, they inevitably entail long wire lengths. Nevertheless, they were the basis of some of the most successful early MPPs: the nCUBE10 with 1024 processors, the nCUBE-2 with up to 8192 processors, the Mark-n hypercubes from JPL, as well as the Intel iPSC-1, iPSC-2, and iPSC-860. They still show up in some NUMA architectures, notably the SGI Origin series.

One-, two-, and three-dimensional meshes are often chosen for interconnects. All meshes involve N/P network switches for N nodes, where P is the number of nodes connected into a single network router. Most commonly, $P = 1$. One-dimensional meshes or rings have only two advantages: they are extremely simple and they involve only a few wires that are short. They require the greatest bandwidth per flop of any interconnect scheme; and they are guaranteed to face contention issues. They also have the greatest diameter of any network. Today, they are virtually unused.

Two-dimensional meshes provide a lower diameter than do rings. (The diameter of a d-dimensional mesh is $O(N^{1/d})$.) They also reduce contention and have relatively few, short wires. [The number of wires in a d-dimensional mesh is essentially $(d \cdot N)$.] Their biggest disadvantage is that they require both relatively high bandwidth per flop and have significant contention, because much of the traffic in simulations of three-dimensional problems will be non-local.

Three-dimensional meshes have numerous advantages: their bandwidth requirements per flop are similar to those of a fat-tree; for algorithms with a great deal of locality of communications, however, they are perhaps lower than those of a fat-tree; they only require short wires; their dimension means that most communications in three-dimensional simulations will be nearest neighbor; their diameter grows as $N^{1/3}$, which for reasonably sized systems is not that different from the $\log_k(N)$ for a fat-tree, where k is the degree of the tree (typically 4). The number of switches required is actually less than that for fat trees if N is large, since N becomes less than $(N/P)\ \log_p(N)$ for $N > P^P$. Wiring is a particular advantage for meshes over all other kinds of interconnects. As described in Section 3, Red Storm uses a three-dimensional mesh interconnect.

2.5.1. Balance requirements

Our experience indicates that taking into account network topology, a well-balanced, 10 000-processor system with a three-dimensional mesh interconnect and uniprocessor nodes will require about 1 byte s^{-1} of processor-to-processor interconnect bandwidth for each flop of processing speed in order to achieve high scalability across the spectrum of our applications. This requirement is not that different if we were to utilize fat-tree interconnects. It is somewhat lower (perhaps as low as 1/2 byte s^{-1} per flop) if we were to utilize a full crossbar interconnect. However, such a crossbar would be prohibitively expensive for Red Storm.

This issue should not be understated. If a parallel system has a parallel efficiency of 90% on large jobs, then doubling the communications speed only provides a 5% gain in performance. By contrast, doubling the processor speed will provide an 80% gain in performance. On the other hand, if the system on that application has a 10% parallel efficiency, then doubling communications speed provides an 80% gain in performance while doubling the processor speed provides only a 5% gain in performance. So, it is important to understand where a design is on the efficiency curve. For most current generation supercomputers, at the 10 000 processor level, typical parallel efficiencies will be 10% or less on many important applications. The Cray T3E, the Earth Simulator and the ASCI Red system are the notable exceptions to this rule.

Let us look at balance in modern supercomputers. In Table II, we compare the architectural balance of several designs produced within the last several years. For comparison, we include the design goals for Red Storm.

In this table, note that the balance ratio is the ratio of peak processing speed of a node to link bandwidth in the interconnect. For ASCI Red and ASCI Blue Pacific, the practical bandwidth limitation is not the link bandwidth but rather the bus bandwidth within the node. This is noted by giving the reduced values in parentheses. For ASCI Blue Mountain and ASCI Q, there are two sets of entries.

Table II. Comparison of balance of Red Storm to existing systems.

Machine	Node speed rating (Mflops)	Link bandwidth (MB s^{-1})	Balance ratio (bytes/flop)
T3E	1200	1200	1.0
ASCI Red[*]	400	800 (533)	2.0 (1.33)
ASCI Red[†]	666	800 (533)	1.2 (0.67)
ASCI Blue Mountain[‡]	500	800	1.6
ASCI Blue Mountain[§]	64 000	1200	0.02
ASCI Blue Pacific	2650	300 (132)	0.11 (0.05)
ASCI Q[‡]	2500	650	0.26
ASCI Q[§]	10 000	400	0.05
ASCI White	24 000	2000	0.083
Earth Simulator	64 000	32 000	0.5
ASCI Red Storm[¶]	4000	6400	1.6

[*]ASCI Red before processor upgrade.
[†]ASCI Red after processor upgrade from 200 to 333 MHz.
[‡]ASCI Blue Mountain and Q: intra-node balance.
[§]ASCI Blue Mountain and Q: inter-node balance.
[¶]From ASCI Red Storm Design Requirements.

For both cases of these shared-memory systems, the intra-node network balance is shown first and then the inter-node balance.

To see the importance of balance, consider the following Amdahl's Law-type considerations. First, Amdahl's Law for parallel computers states that the speedup achievable for a problem running on N processors over the same problem running on one processor is given by

$$S_A = [1 + f_s]/[1/N + f_s]$$

where f_s is the serial fraction of the workload. Now, to achieve a parallel efficiency of 80% on 10 000 processors, Amdahl's Law implies that the serial fraction must be less than 0.000 025. Contrary to the expectations of many in the community prior to the work at Sandia on scaled speedup [23], applications often achieve this kind of efficiency. Now Amdahl's Law understates the problem. A more limiting factor in parallel efficiency is overhead due to communications. To a good approximation, we may calculate the real speedup as

$$S = [S_A]/[1 + f_c R_{p/c}]$$

where f_c is the fraction of work devoted to communications overhead and $R_{p/c}$ is the ratio of processor speed to communications speed (the inverse of the balance ratio from above). In Table III, we show the effect of communications balance on four hypothetical applications on two hypothetical machines. The two machines are presumed identical, except that the first has a balance ratio of 1 and the second a balance ratio of 0.05. These two choices more or less span the range of architectures in Table II. The four applications have communications overhead fractions of 0.001, 0.01, 0.05, and 0.10. In all four cases we assume zero serial work for simplicity.

Table III. Machine efficiency as a function of balance factor and communications overhead fraction.

	Communications fraction			
Balance ratio	0.001	0.01	0.05	0.10
1.0	0.999	0.99	0.95	0.91
0.05	0.98	0.83	0.50	0.33

In Table III, we see that if the workload is most highly compute bound or embarrassingly parallel, it is not necessary to invest in a highly balanced network. If, however, a large fraction of the workload has significant communications overhead, then it pays to do so. In the not so extreme case that 10% overhead is incurred due to communications, having a poor balance in the network could lead to the effective loss of the majority of the system's capability. For these reasons, we have chosen to provide a highly balanced network in ASCI Red Storm.

2.6. Software issues

Our software architecture is examined extensively in Section 4 below. Here, we simply note the choices that we considered and the design decisions we arrived at. As noted above in this section, reliability and scalability are strongly dependent on software design and execution. Because of cost and maintainability considerations, we wanted to leverage as much existing open-source software as was feasible and wise in the design of Red Storm. One choice—in dealing with operating systems—was whether to utilize a proprietary UNIX or an open-source UNIX, either BSD or Linux. Because we had extensive internal experience with Linux, and because the known shortcomings of Linux would not come into serious play (due to the limited manner in which we would employ it on Red Storm), we chose Linux. We then faced the issue of whether to deploy Linux throughout the system, as is done on clusters such as Cplant$^{\text{TM}}$, or to limit the function of Linux to only those partitions of the system that would depend critically on full UNIX services. As noted above, we chose to limit Linux to the service and I/O partitions, and to rely on a lightweight kernel (LWK) operating system, Catamount, on the compute nodes. Catamount is the third generation of lightweight kernels developed at Sandia in partnership with the University of New Mexico. It is a simple port of the Puma operating system [15] that we developed for the Intel Paragon and re-deployed as Cougar on the ASCI Red system. Of course, Cougar had to be modified to meet the requirements of the new architecture. However, the LWK design philosophy is nearly identical to that employed on our previous machines. It would have been desirable to enhance or re-design portions of the operation system environment, but time and budget limitations prevented us from making any significant modifications to Red Storm.

One area in which we decided to make extensive software changes in moving from ASCI Red to Red Storm was in the area of parallel file systems. I/O has been the Achilles' heel of nearly every MPP built. It was fairly straightforward to create a very powerful hardware system to support I/O. In contrast, making a full-featured, high-performance file system to take full advantage of that hardware was beyond our means. We had to choose to leverage ongoing open-source efforts. It is important to understand the challenges facing a designer wanting to get full features and high performance in a parallel file system for an MPP on the scale of Red Storm. At any given time there will be jobs from tens of users on the system. These jobs will compete for I/O resources in an asynchronous fashion. If we think of the compute nodes generically as parallel clients and the I/O nodes as servers, we can have many parallel clients requesting I/O services simultaneously. In such a situation, load imbalance between I/O nodes and compute nodes and among I/O nodes could destroy efficiency. Well-designed applications will attempt to stage I/O internally so as to avoid self-contention. However, many application developers may not have expended the same degree of rigor in developing I/O strategies as they have in parallelization techniques. In addition, individual code designers cannot mitigate the problems caused by asynchronous contention for resources among different jobs. These issues put an enormous burden on the parallel file system. It is critical to size the service and I/O partitions adequately to handle the expected I/O traffic. Otherwise I/O is fully capable of becoming the serialization bottleneck that reduces the performance of the system unacceptably. From an architecture point of view, I/O and parallel file systems are among the biggest risks we face in developing a balanced architecture.

3. THE RED STORM COMPUTER SYSTEM

The following sections describe the goals and requirements of the Red Storm architecture and provide details on the hardware components of the system.

3.1. Red Storm architectural goals

As mentioned above, Sandia has a long history in MPP computing, starting with the first 1024 processor nCUBE-10 computer system that was installed at Sandia in 1987. Since then, Sandia has had two 1024-processor, second-generation nCUBE-2 MPPs, a 16K processor Thinking Machine CM2, a 3600+ processor Intel Paragon, and Sandia's current supercomputer, the Intel ASCI Red machine with over 9500 processors. Sandia also has over 2500 processors in the Cplant^{TM} cluster. Over this period, computing technology has changed dramatically. However, the basic characteristics that made the first nCUBE-10 a highly scalable machine and ASCI Red highly scalable are the same. These are a communications network that provides a high computation to communication ratio, highly scalable operating system and system software, an integrated system design, and a level of reliability that allows for meaningful work to be accomplished between interrupts.

The following goals for future large parallel computer system architecture and performance were developed based on Sandia's extensive experience in MPP computing. The Red Storm architecture and performance requirements are based on these goals. The major goals are as follows.

1. *Balanced system performance*. The balance between processor, memory, interconnect, and I/O performance needs to provide good overall performance on a broad set of scientific and engineering application codes. These application codes need to achieve good parallel efficiency at the level of 10 000 or more processors.

2. *Usability*. The functionality of system hardware and software must meet the needs of users for very large scale MPP computing, and not general purpose computing.
3. *Scalability*. System hardware, software, and performance scale from a single cabinet system to a system with approximately 30 000 processors. The physical system size and equipment part count need to scale linearly with the computational power of the system.
4. *Reliability*. The machine needs to stay up long enough between interrupts to make real progress on completing application runs, at least 50 h mean time between interrupts (MTBI). Sandia's application codes may run for a hundred or more hours on a single problem. For these kinds of applications, the measure of a supercomputer's reliability is MTBI of the application code and not percent availability. To reiterate, it is possible to have a machine with 99% availability that is nearly useless for our workload because it has a high application interrupt rate.
5. *Upgradeability*. System performance needs to be able to be upgraded with a processor swap or board swap and additional cabinets to a factor of three or more in capability. Supercomputers are expensive, and it is highly desirable to be able to extend the life of the machine with relative ease through an upgrade, rather than through complete replacement.
6. *Red/Black switching*. It is important for Sandia to have the ability to switch major portions of its large capability supercomputers between classified and unclassified computing environments. Most code development, and a significant fraction of Sandia's work, can be done in the unclassified environment; however, many calculations must be done in the classified environment. Red/Black switching of major sections of the machine between environments allows Sandia to have access to its largest capability computing resource in both environments, although not simultaneously.
7. *Space, power, cooling*. Leading edge capability supercomputers are very large, and require a significant amount of power and cooling. Dense packaging and careful design can substantially reduce the system physical size and the power and cooling needed to operate it.
8. *Price/performance*. Excellent performance per dollar is possible through the use of high-volume commodity parts where feasible.

All of these goals except for Red/Black switching are generally applicable to large parallel supercomputers.

3.2. Red Storm architecture and performance requirements

The Red Storm architecture requirements were designed to produce a system that meets the above goals. The major architectural requirements for Red Storm are as follows.

1. *Red Storm is a tightly coupled MPP that is designed to be a single system and not a cluster.* This requirement is designed to help meet the goal of system balance and to avoid many of the scaling issues that have plagued large clusters.
2. *Red Storm has a fully connected three-dimensional mesh interconnect in which each processor has one bi-directional connection to the primary communication network.* This requirement addresses system scalability and upgradeability. Mesh interconnects grow linearly in size with the number of nodes. As the system size increases the complexity of a mesh interconnect does

not increase, although, the number of hops to get from one end of the mesh to another does increase. The maximum size of Red Storm is limited by the number of entries supported in the interconnect router tables. This limit is 32 000.

3. *Red Storm has three functional hardware partitions: (1) service and I/O, (2) compute, and (3) reliability, availability, and serviceability (RAS) and system management*. This requirement addresses system balance and overall performance by requiring dedicated hardware for application code support, application code execution, and system management and monitoring.
4. *The Red Storm compute node partition is divided into three sections, and the service and I/O node partition, and the RAS and system management partitions are divided into two sections each to support Red/Black switching*. This requirement provides Sandia with the flexibility to operate Red Storm in multiple classified and unclassified configurations while always providing the users with access to their data.
5. *The Red Storm operating system (OS) is partitioned to match the hardware partitioning with a full UNIX or UNIX-like OS on the service and I/O nodes, a lightweight kernel (LWK) on the compute nodes, and a real-time or real-time like OS on the RAS nodes with a full UNIX or UNIX-like OS on the system management workstations*. The user interface is a full UNIX while the compute node OS is a LWK which provides minimal OS overhead and optimal application code performance. The LWK requirements preclude demand paging and a number of other functions that would negatively impact system performance. Using an LWK on the compute nodes also increases the overall system reliability as the LWK has far less code paths that can potentially fail than a full OS.
6. *The Red Storm disk storage system is divided into two sections to match the service and I/O nodes and to support both classified and unclassified computing*. This requirement is necessary for Red/Black switching.
7. *Red Storm compute nodes are diskless and perform all disk storage and external network I/O through the service and I/O nodes*. This requirement is necessary for Red/Black switching. However, it also exists to keep the compute node OS simple and to increase system reliability by not having a very large number ($>10\,000$) of unprotected disks spread throughout the system.
8. *All disk storage in Red Storm is RAID 3 or RAID 5. Each RAID has parity and its own hot spare disk*. These requirements address system reliability.
9. *Red Storm has an independent RAS and system management network that is designed to monitor the system and provide a path for system management without interfering with application code performance*. For a system of the size of Red Storm a comprehensive RAS system is absolutely necessary for maintenance and reliability.
10. *Red Storm is required to have an MTBI of 50 h for application codes, where the interrupt is caused by a failure in system hardware or software. Further, Red Storm is required to have an MTBI of 100 h for rebooting the system because of a failure in system hardware or software*. The length of time between interrupts is the best indicator of system reliability for supercomputers. These machines need to run very large problems for very long times.
11. *Red Storm is built from high-volume Commodity Off The Shelf (COTS) parts wherever feasible. Red Storm has a limited number of custom parts*. In addition to the network interface/router chip, which was designed specifically for Red Storm, the CPU boards and the packaging are unique to the machine. The rest of the parts in the system are either high-volume commodity parts or at least standard production items.

The Red Storm performance requirements, together with the architectural requirements, result in a computer system that meets the above goals. The performance requirements were designed to produce a computer system with balanced performance. The major performance requirements are as follows.

1. *Low-latency interconnect*. Nearest-neighbor MPI latency as measured by a ping/pong test divided by 2 is less than 2.0 μs.
2. *High-bandwidth interconnect*. Each link of the three-dimensional interconnect has a peak bandwidth greater than 1.5 bytes/flop (peak) of the compute processor.
3. *High minimum bi-section bandwidth*. The minimum bi-section bandwidth is the bandwidth through the plane of the three-dimensional compute node mesh that has the least bandwidth. The minimum bi-section bandwidth is at least 0.05 bytes/flop (peak) of the full machine.
4. *Sustained high I/O bandwidth*. The sustained bandwidth requirement for I/O in Red Storm is $\sim$0.0025 bytes/flop (peak) of the machine.
5. *High external network bandwidth*. A sustained external network bandwidth from/to Red Storm of 0.001 25 bytes/flop.
6. *Scaled real performance*. Performance that scales linearly or nearly so with problem size for Sandia's application codes.

3.3. The Sandia Red Storm computer system

The Red Storm computer system is under development by Cray Inc. The first Red Storm machine is being built for Sandia and is expected to be operational at Sandia in Albuquerque, New Mexico, in August 2004.

3.3.1. Physical layout

The Sandia Red Storm supercomputer will have 124 CPU cabinets and 16 Red/Black switch cabinets. The 140 cabinets are laid out in four rows of 35 cabinets each. This cabinet layout is shown in Figure 1 below. Out of the 124 CPU cabinets, 108 cabinets will have compute nodes and 16 cabinets will have service and I/O nodes. The service and I/O node cabinets are equally split between the classified and unclassified partitions with eight on each end of the machine. The compute node cabinets are split into three sections with seven cabinets in each row normally unclassified and seven cabinets in each row normally classified. (In Figure 1, the normally classified cabinets are shown in red and the normally unclassified cabinets are shown in black.) The other 13 cabinets in each row will be switched between the unclassified environment and the classified environment as needed. (These cabinets are shown as white in Figure 1.) In addition, under special circumstances, all of the compute nodes may be switched to the classified or unclassified environments.

The eight service and I/O node cabinets on each end of the machine are divided equally among the four rows, two cabinets per row. These cabinets will not be switched between the classified and unclassified environments. All disk storage connections and all external network connections to the machine are through these cabinets.

The Red/Black switch cabinets, four cabinets per row, provide for making or breaking a connection between Y and Z planes in the high-performance machine interconnect. Except for cable connections, these cabinets are empty. They do not require power or cooling.

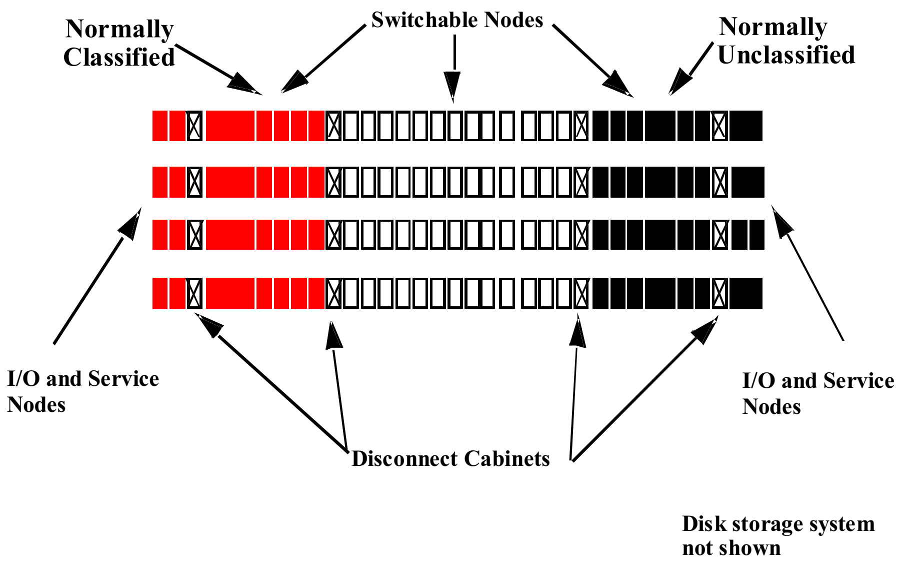


Figure 1. Sandia Red Storm system cabinet layout.

All CPU and Red/Black switch cabinets are the same physical dimensions, 24″ W × 48″ D × 80″ H. The total floor space required for the machine, including 48″ of access on all sides and in all isles, is 2808 ft^2. When the disk storage cabinets are included the total machine size is approximately 3000 ft^2.

3.3.2. CPU cabinet configuration

Each compute node cabinet has three card cages. Each card cage has eight compute node boards, and each compute node board has four AMD Opteron processors and four high-performance NIC/router chips. In addition, each board has a RAS processor and a number of Voltage Regulator Modules (VRMs). Each Opteron processor has four memory slots for DDR DIMMs. As shown in Figure 1, the node boards are oriented vertically and the direction of airflow through the cabinet is from bottom to top. Each card cage has a passive back plane that provides cable connections for connecting card cages together and internal connections for the boards within the card cage.

The service and I/O node cabinets are very similar to the compute node cabinets. The same card cage and backplane are used, and there are eight boards per card cage. However, each service and I/O node board has two AMD Opteron processors and four NIC/router chips. These boards also have four DDR DIMM slots per processor. In addition, they have PCI-X slots for disk storage connections and for external network connections.

3.3.3. Topology

The compute nodes are configured in a $27 \times 16 \times 24$ (X, Y, Z) three-dimensional mesh, and the service and I/O nodes are configured in a $2 \times 8 \times 16$ (X, Y, Z) three-dimensional mesh. There are 16×24 (Y, Z) or 384 links in the mesh at each Y, Z plane. Of these, 128 (8×16) are connected on the interface between the service and I/O nodes and the compute nodes. The minimum bi-section is 16×24 (Y, Z) for the compute nodes and 2×8 (X, Y) for the service and I/O nodes. At each point in the mesh, service and I/O node and compute node, there is a high-performance router.

Each router has seven bi-directional ports. Six of the ports are used to make the three-dimensional mesh and the seventh port connects to the NIC. The NIC connects to the AMD Opteron through a HyperTransport [21] link.

The 108 compute node cabinets have a total of 10 368 single processor nodes. Each compute node cabinet has 96 nodes per processor configured in a two-dimensional mesh of 4×24 (Y, Z). The eight service and I/O node cabinets on each end of the machine have a total of 256 single processor nodes. Within each service and I/O node cabinet there is a two-dimensional mesh of 2×16 (Y, Z).

3.3.4. RAS and system management design

Red Storm has a comprehensive RAS and system management capability. The classified and unclassified environments each have a primary Red Storm Management Workstation (RSMW) and an online backup. These workstations are connected to the Red Storm system cabinets through a tree-structured, switched, and dedicated Ethernet.

Each Red Storm cabinet has a RAS processor (L1) and each Red Storm CPU board has a RAS processor (L0). The cabinet L1 processor controls power-on of the CPU cabinet and it monitors the L0 processors. The L1 also acts as a concentrator for L0 messages that go to the RSMW. The L1 processor is connected to the cabinet L0 processors through a dedicated Ethernet.

The L0 processor on each board performs most of the monitoring functions, including checking for the node heartbeat, and it provides a link to the Opteron processors and the NIC/router chips. JTAG is also implemented on each board to provide hardware monitoring through the L0 processor and with direct connection to the board.

3.3.5. System software

Red Storm has three operating systems. The service and I/O nodes run a full version of the Linux operating system that has been functionally optimized for various service and I/O node functions such as I/O and user support. As mentioned above, the compute nodes run a LWK named Catamount. The RAS nodes run a stripped down version of Linux, and the RSMW runs a full version of Linux.

Details of the high-performance system software architecture used on compute nodes, including the high-performance message-passing layer and scalable job loader, are provided in Section 4 below. The Red Storm runtime environment also provides users with a batch scheduler (PBS), a compute node allocator, and libraries for MPI, I/O, and math. The compute node allocator tries to optimize job placement based on the topology of the network, but will assign jobs any available nodes if necessary to accommodate the job within scheduling constraints.

Red Storm has two file systems available to users, a UNIX File System (UFS) and a parallel file system. UFS is a serial file system that is implemented on a RAID. The parallel file system is designed to provide striping of files across many RAIDs and to provide very high-performance I/O. The specific parallel file system that will be provided on Red Storm has not been determined at this time. Several open-source and commercial parallel file systems are under evaluation.

The Red Storm software environment includes a number of tools. These include compilers, a parallel debugger, and a performance monitor. Compilers for Fortran, C, and C++ are included. In addition to compiling on Red Storm, users will have the option of compiling their codes for Red Storm using a cross-compile environment on an Opteron workstation. The parallel debugger included in the Red Storm tools is TotalView. TotalView has become a de-facto standard for ASCI machines. The performance-monitoring tool has yet to be selected. It will provide a user interface for counting a wide variety of operations in the Opteron processor.

A Graphical User Interface (GUI) tool is being developed for displaying system management and monitoring data on the RSMW. Also, an accounting package from another Cray product is being ported to Red Storm.

3.3.6. Performance

Sandia's current supercomputer, ASCI Red, has proven to have very good application scalability on a broad set of scientific and engineering problems. Red Storm will be better balanced than ASCI Red. Red Storm will have a better node memory system, a higher performance interconnect, and a much higher performance I/O system. The memory bandwidth per peak floating point operation per second is higher and the latency to local node memory is significantly lower than on ASCI Red.

The Red Storm interconnect will have higher bandwidth links and more of them than in ASCI Red for a similar size system. Even relative to the peak performance of its nodes, Red Storm's link bandwidth will be higher than ASCI Red's. The minimum bi-section bandwidth in Red Storm will be substantially higher than in ASCI Red because of the higher bandwidth links and because the minimum bi-section plane has six times as many links. In addition, because of increased processor performance and a smart NIC, MPI latency will be around a factor of seven lower in Red Storm than in ASCI Red.

Disk I/O bandwidth on Red Storm will be approximately a factor of 50 greater than it is on ASCI Red. For some applications, I/O performance is a limiting factor in the overall performance of ASCI Red.

A detailed comparison of ASCI Red and Red Storm is given in Table IV.

The Red Storm computer system will be operational at Sandia in August 2004. It will have a peak performance of a little over 40 Tflops. It will have more than 55 TB s^{-1} memory bandwidth. The Red Storm interconnect will have over 120 TB s^{-1} of interconnect bandwidth. Each end of Red Storm, classified and unclassified, will be able to sustain 50 GB s^{-1} of disk I/O bandwidth and 25 GB s^{-1} of external network bandwidth. Sandia expects to achieve more than 25 Tflops on the MP-Linpack benchmark. More importantly, Sandia expects to achieve excellent parallel efficiency on very large ASCI problems running on the full machine, all 10 368 compute nodes.

4. SANDIA'S HIGH-PERFORMANCE COMPUTING SOFTWARE ARCHITECTURE

In this section, we describe our architecture for scalable, high-performance system software. The system software architecture that we have developed is a vital component of a complete system.

Table IV. Comparison of ASCI Red to Red Storm.

	ASCI Red	Red Storm
Full system operational time frame	June 1997 (processor and memory upgrade in 1999)	August 2004
Theoretical peak (Tflops)	3.15	41.47
MP-Linpack performance	2.379	>30 (est.)
Architecture	Distributed memory MIMD	Distributed memory MIMD
Number of compute nodes	4730	10 368
Processors per node	2	1
Number of service and I/O nodes, each end	73	256
Processor	Intel Pentium II @ 333 MHz	AMD Opteron @ 2.0 GHz
Compute node memory	1.2 TB	10.4 TB (eventually expandable to 83 TB)
Memory per compute node	256 MB	1.0 GB
System memory B/W	2.5 TB s^{-1}	55 TB s^{-1}
Disk storage	6.25 TB each end	120 TB each end
Parallel file system B/W	1.0 GB s^{-1} each end	50 GB s^{-1} each end
External network B/W	200 MB s^{-1} each end	25 GB s^{-1} each end
Interconnect topology	3-D Mesh, $38 \times 32 \times 2$ (X, Y, Z)	3-D Mesh, $27 \times 16 \times 24$ (X, Y, Z)
Interconnect performance MPI latency	15 μs 1 hop, 20 μs max	2.0 μs 1 hop, 5.0 μs max
Bi-directional link B/W	800 MB s^{-1}	6.0 GB s^{-1}
Minimum bi-section B/W	51.2 GB s^{-1}	2.3 TB s^{-1}
Full machine RAS system		
RAS network	10 Mbit Ethernet	100 Mbit Ethernet
RAS processors	1 for each 32 CPUs	1 for each 4 CPUs
Operating systems		
Service and I/O nodes	TOS (OSF1)	LINUX
Compute nodes	Cougar (LWK)	Catamount (Cougar)
RAS nodes	VX-Works	LINUX
Red/Black switching (processors in each section)	2260–4940–2260	2688–4992–2688
System foot print	~2500 ft^2	~3000 ft^2
Power/cooling requirement	850 kW	1.7 MW

System software is an important area of optimization that directly impacts application performance and scalability, and one that also has implications beyond performance. System software not only impacts the ability of the machine to deliver performance to applications and allow scaling to the full system size, but also has secondary effects that can impact system reliability and robustness. The following sections present an overview of our system software architecture and provide important details necessary to understand how this architecture impacts performance, scalability, reliability, and usability. We discuss examples of how our architecture addresses each of these areas and present reasons why we have chosen this specialized approach. This section concludes with a discussion of the specifics of the implementation of this software architecture for Red Storm.

4.1. Target system architecture

Our target system architecture partitions the nodes of a massively parallel system based on functional considerations. We partition the nodes into three natural sets: compute nodes, service nodes, and I/O nodes. More generally, we can envision having other partitions, such as database partitions and visualization partitions. In the context of system software design, the primary advantage of partitioning is that it allows the system software to be tailored to specific needs. See [24] for a more complete description of functional partitioning.

The compute partition is dedicated to delivering resources to parallel application processes. The needs of applications and the usage model of the machine largely determine the functional requirements of compute nodes. We view parallel applications as being resource constrained, such that they can scale to consume all of at least one type of resource (e.g. memory, memory bandwidth, processing, network bandwidth, etc.) provided by the system. In considering these applications, the primary concern is reducing execution time. A single run of a resource-constrained application may use the entire system for several days. In our architecture, we restrict the functionality that is locally available on compute nodes to the absolute minimum that is required by our important applications. The usage model of the machine, which emphasizes reducing execution time for one or a few parallel applications, allows us to employ a space-sharing model for the compute nodes. In this model, the allocator binds a group of compute nodes to a single user. We also assume that the compute partition is homogeneous.

The service partition provides access to the compute partition. As a minimum, nodes in the service partition support user logins (authentication) and parallel application launch. More typically, these nodes also perform a variety of other activities such as compiling codes, editing files, sending email, and checking the status of the nodes and jobs in the compute partition. The functions of the service partition are, in general, those functions of a workstation, which, when moved into the high-performance realm, involve the interaction of serial, user-directed interfaces with parallel application codes. In our architecture, these service functions are decomposed into an interface component that runs in the service partition and a scalability component that runs in the compute partition. Most interface components run as sequential programs on a single service node. The I/O partition provides access to a global parallel file system, and may also contain nodes that provide access to secondary storage systems or high-performance network interfaces to other systems.

4.2. The Puma/Cougar operating system

Puma [15] is the second-generation lightweight compute-node kernel designed and developed by Sandia and the University of New Mexico. Intel, with help from Sandia, ported Puma from the Intel Paragon to ASCI Red, at which time Intel productized it as Cougar. Sandia continued development of Puma as a research project while Intel continued to develop Cougar specifically for ASCI Red. While there are subtle differences between the implementations of the two, we consider them to be identical in the context of this paper, and for simplicity, we will refer to Cougar henceforth. The following provides an overview of the important components and characteristics of Cougar, including our high-performance data movement layer called Portals.

Cougar consists of a Quintessential Kernel (QK) and a Process Control Thread (PCT). The QK is the lowest level of the operating system. It sits on top of the hardware and performs hardware services on behalf of the PCT and user-level processes. The QK supports a small set of tasks that require execution in privileged supervisor mode, including servicing network requests, interrupt handling, and fault handling. It also fulfills privileged requests made by the PCT, including running processes, context switching, virtual address translation and validation. However, the QK does not manage the resources on a compute node. It simply provides the necessary mechanisms to enforce policies established by the PCT and to perform specific tasks that must be executed in supervisor mode.

The PCT is a privileged user-level process that performs functions traditionally associated with an operating system. It has read/write access to all memory in user space and is in charge of managing all operating system resources. This involves process loading, job scheduling, and memory management. While QKs do not communicate with each other, the PCTs on the nodes that have been allocated to a parallel application communicate to start, manage, and tear down the job.

The PCT and the QK work together to provide a complete operating system. The PCT will decide what physical memory and virtual addresses a new process is to have and at the behest of the PCT, the QK will set up the virtual addressing structures for the new process that are required by the hardware. The PCT will decide which process is to run next and at the behest of the PCT, the QK will flush caches, set up the hardware registers, and run this process. There is a clear separation between resource management and kernel task execution. The PCT is responsible for setting the policies and the QK is responsible for enforcing them.

One of the important characteristics of Cougar is that it is based on multiple levels of trust. A QK trusts other QKs as well as the other operating systems that run in the other partitions of the machine. This also implies that all of the operating systems trust the network, since messages can only be placed on the network by another operating system. QKs do not trust PCTs or user-level processes. The PCT trusts the QK and other PCTs. PCTs do not trust user-level processes. User-level processes trust both the QK and the PCT. This trust model is necessary to achieve high performance from the network. Since the network is trusted, privileged information in a message, such as the source of the message, does not need extensive verification.

4.2.1. Processor modes

In this section, we describe the different ways in which Cougar can manage multiple processors on a compute node. In Cougar, a single processor, the system processor, manages all of the resources on a node. This is the only processor that performs any significant processing in supervisor mode. The remaining processors, if any, run application code and only rarely enter supervisor mode. These processors are called user processors. For the sake of this discussion, we assume that a node has only two processors and that there is a single user process. See [25] for a more detailed description of the implementation of these different modes.

The simplest mode of operation is to run the kernel and the user-level process on the system processor and simply ignore the user processor. This mode is colloquially referred to as heater mode, since the user processor only generates heat. System calls from a user process are initiated by a trap instruction to the kernel, which handles the request and re-establishes the context of the user process. This mode does not provide any significant performance advantages, but it is the simplest mode to make operational and has historically been the default mode.

The kernel co-processor mode runs the kernel on the system processor and the user-level process on the user processor. In this mode, the QK polls the external devices and looks for system call requests from the user-level process. Since the time to transition between user mode and kernel mode can be significant, this mode offers the possibility of increased performance for handling system calls and servicing devices. This mode is also sometimes referred to as message co-processor mode, since the kernel is able to dedicate a processor to servicing the network.

In user co-processor mode, the kernel and the user process run on both processors. However, the kernel and user codes that run on the user processor do so in a very limited way. The kernel code running on the user processor does not perform any management activities. It simply notifies the system processor of requests. The user code that runs on the user processor must run within the same context as the process on the system processor and is limited in the system calls that it can make. Access to the user processor from within an application is via an interface for running co-routines. Because of this non-standard interface, most application programs rarely use this mode, and those that do, use a specialized version of a standard math library.

Finally, virtual node mode supports running the kernel and a user process on the system process and a separate user process on the user process. As the name implies, this mode treats each processor as a separate node. The available memory on a node is divided in half and two separate user address spaces are created. All kernel services are fully supported on both processors, but the kernel runs only on the system processor. There is no support for shared memory, so data transfers between the two processes on a node must be done via the network. This mode has the advantage of using all of the processors on a node in a manner that is transparent to the application code or user.

The user chooses the mode in which a parallel application runs when the job is started. The application co-processor mode is the only mode that requires an appropriate library to be linked into the application before it is run. The user can easily switch between heater mode, kernel co-processor mode, and virtual node mode without any modifications to the application simply by specifying the desired mode to the parallel job launcher.

4.2.2. Portals

Message passing performance is a critical aspect of massively parallel, distributed memory machines. Even a single memory copy in the network stack can severely impact performance. For this reason, we have designed a flexible communication mechanism that allows data transfers directly from user memory on one node to user memory on another. We call this mechanism Portals. The following provides an overview of the important details of Portals in Cougar. See [15] for a more complete discussion.

One of the important features of Portals in Cougar is that all of the structures associated with message passing are in user space. The kernel is only responsible for traversing these data structures and depositing messages into user space based on the content of the structures. The kernel does not enforce any specific protocol or provide any buffering of messages. This allows the kernel to remain at a fixed size regardless of the size of the parallel job or the amount of network resources a job requires.

All protocols are implemented at the user level. Because of this, Portals must be flexible enough to support a wide variety of protocols. Parallel application message passing libraries, like MPI [12], as well as I/O protocols, and communication between the PCTs all must use Portals.

A Portal is referenced through an index into a Portal table, where each table entry refers to either a match list or a memory descriptor. A memory descriptor describes the layout of the memory associated with the Portal. Matching lists provide Portals with a matching semantic that can be used to bind specific messages to specific memory regions. Each entry of the matching list has a memory descriptor associated with it. A Portal may refer to a memory descriptor directly or to multiple memory descriptors indirectly through a matching list. The Portal table, matching lists, and memory descriptors all reside in user space.

An important feature of Portals is the ability to deliver messages without the direct involvement of the user-level process. Once the Portal structures have been put in place to receive a message, a process need not poll the network in order for the data transfer to complete. The process need not even be running for the kernel to deposit the message directly into the process' memory. This capability, in conjunction with kernel co-processor mode, provides the ability to fully overlap computation and communication, even for somewhat complex protocols like those required by an MPI implementation. Portals are also connectionless, which eliminates the overhead associated with connection establishment and also helps to reduce the amount of state that is needed for message passing.

4.2.3. Parallel runtime system

An additional important component in our system software architecture is the parallel application launcher. Since we have removed many of the services of a traditional operating system from our compute node kernel, the runtime system plays an integral role in providing services to parallel applications. In our environment, the parallel job launcher not only interacts with the PCTs on the compute nodes to start the job, it provides services to the application while it is running. This section provides an overview of our parallel runtime system.

One of the challenges in any massively parallel processing system is providing a parallel runtime environment that allows for fast startup of parallel jobs. Since our primary goal is to reduce the amount of time required to achieve a solution, it is critical for a large-scale parallel machine to start jobs as efficiently as possible. While most suppliers of large-scale parallel computing platforms emphasize delivering performance to an application once it is running, few address the time spent getting the application started. On many systems, this time can be significant.

The parallel job launcher component of our runtime system is called yod. Yod contacts a compute node allocator to obtain a set of compute nodes, and then communicates with a primary PCT to move the user's environment and executable out to the compute nodes. The primary PCT works with the secondary PCTs in the job to efficiently distribute these data to all of the compute nodes participating in the job.

Once a job has started, yod serves as an I/O proxy for all standard I/O functions. Compute node applications are linked with a library that redefines the standard I/O library routines and some system calls. This library implements a remote procedure call interface to yod, which actually performs the operation locally and then sends the result to the compute node process. Yod also disseminates some UNIX signals that it receives out to the processes running in the parallel job. When yod receives a signal, it sends a message to the primary PCT, which distributes the message to the other PCTs in the job and delivers the desired signal to the application process. This feature can be very useful for operations such as user-level checkpointing and killing jobs.

4.3. Reasons for a specialized approach

We have chosen a specialized approach to system software rather than a commodity-based approach. Our early experiences with the limitations of the OSF/1AD [26] operating system on the Intel Paragon led to the design and development of our own system software, namely the SUNMOS [14] and Puma [15] lightweight kernels. As we discussed above, these lightweight kernels are an important part of a more complete approach to system software that involves the operating system, network communication stack, and parallel runtime system. Simply stated, our approach to system software is an optimization based on the hardware architecture, the programming model, the usage model of the machine, and the requirements of large-scale parallel applications. The following sections discuss these optimizations in more detail.

4.3.1. Optimizing compute node resources

In this section we describe the features of our software architecture that are aimed specifically at maximizing the resources delivered to parallel application processes.

Maximizing the amount of CPU time delivered to a parallel application process is critical to achieving high performance and scalability. Our lightweight kernel was designed to minimize the amount of processing the operating system takes away from the application process.

Unlike traditional full-featured operating systems based on UNIX, our lightweight kernel does not continuously take timer interrupts to analyze the state of the node. For example, a Linux 2.4 kernel running on an Alpha processor takes an interrupt every millisecond to assess the state of the machine and perform housekeeping activities. Our kernel and scheduler are designed to avoid such spurious activities and only require processing to perform critical functions. Our environment does not support running daemon processes on compute nodes, so only the QK, PCT, and application compete for the CPU(s).

The ability to choose the way in which multiple processors on a node are used can have a great impact on performance. For codes that are memory bandwidth or network bandwidth limited, the application may benefit from running in kernel co-processor mode rather than application co-processor mode. The user is given explicit control and can determine which method is best.

Cougar supports virtual addressing to provide memory protection between all of the processes on a node. However, it does not provide demand-paged virtual memory. Since our nodes are diskless, paging would be prohibitively expensive and would interfere with the activities of other nodes using the same network paths and disks. In our experience, most well designed parallel applications avoid paging in environments where it is supported. Regardless, applications are much better at determining which memory pages are not needed anymore. These pages can be filled with more data from disk. Taking advantage of high-performance I/O and network access is much more efficient than a general memory page replacement strategy implemented in the operating system.

Our architecture is also designed to maximize the amount of memory resources delivered to parallel applications. By definition, our lightweight kernels are meant to minimize the footprint of the operating system. Both SUNMOS and Cougar consumed less than 1% of the memory on even a small-memory compute node, in contrast to UNIX-based operating systems that can consume several megabytes. Memory footprint was critical on the Paragon where OSF/1-AD consumed more than half of the available memory on a node. The trend toward larger memory capacity on compute nodes has somewhat lessened the importance of a small footprint. However, we still consider memory usage in the context of the entire system, where wasting one or two megabytes per compute node ends up wasting tens of gigabytes of memory on the machine.

We also maximize the amount of memory given to an application by providing a connectionless networking layer that relies on structures in user space. The size of Cougar is fixed. It is not dependent on the size of the parallel application and does not change with the amount of message passing that an application performs. The application is given control over how much of its memory is to be used for buffering messages.

In addition to memory size, our architecture also attempts to maximize memory bandwidth delivered to applications. The Cougar kernel is designed to use a larger memory page to avoid the overhead of translation lookaside buffer (TLB) management. Larger memory pages mean reducing TLB flushes that can significantly degrade memory performance. Most traditional UNIX operating systems use the smallest memory page that is supported by the processor in order to implement virtual memory paging efficiently.

Our architecture is also intended to maximize network performance. The QK and Portals have been designed to minimize message passing latency and provide the full bandwidth of the network. Since all message passing structures are in user space, they can be manipulated directly by the application without going through the kernel. Cougar uses a physically contiguous memory model where virtual addresses map directly to physical addresses. In this model, translation and validation of user virtual addresses is done through a simple offset and bounds check calculation. And since Cougar does not support virtual memory paging, there is no need to insure that the memory pages used in a network transfer are resident. These characteristics all help to significantly reduce the amount of overhead required by the Cougar to perform a network transfer. In addition, a latency-bound application is likely to use kernel co-processor mode to reduce the impact of context switching on message startup.

There are also design features that are aimed at maximizing network bandwidth. Portals are designed to deliver messages directly into user space without any intermediate memory-to-memory copies. The ability of Portals to overlap computation and communication is intended to allow applications to perform useful work while large data transfers are ongoing. A side effect of our space-sharing model is that processes in a job are gang scheduled by default. This is especially important for tightly coupled applications that compute and communicate at regular intervals. The runtime system also plays a part in attempting to maximize network bandwidth. Our architecture includes an intelligent compute node allocator that understands the topology of the network. An allocator can assign nodes to a job in a way that minimizes the number of network hops between nodes.

Finally, our system software architecture also aims to increase the robustness and reliability of the system. A simple, small, compute node operating system is much easier to make reliable than a large, monolithic, full-featured operating system. One of the reasons for the separation in functionality between the QK and PCT is so that the QK can always be available to service the network. Even when faced with a failure of the PCT, which is equivalent to a traditional operating system crashing, the QK is able to continue to function and service the network. This is an extremely important feature for the Paragon and ASCI/Red machines where failure to service the network at a single node will eventually cause the entire network to lock up. Pushing the complexity of the operating system out to the runtime significantly reduces the complexity of the lightweight kernel.

4.4. Basic principles

There are several basic principles upon which our system software architecture is based. We discuss them here for completeness.

It is necessary to divide the machine into logical partitions based on functionality. Different parts of the system have different functional requirements, and partitioning allows us to tailor the system software on a node to its particular function. Partitioning reduces the conflicts that can occur between performance requirements and features.

There are fundamental differences between parallel systems and distributed systems. Most system software architectures for parallel machines are an attempt to scale distributed computing models. There are fundamental differences in resource acquisition, resource management and functionality. Distributed systems are designed for dynamic environments and coarse-grain parallelism, while parallel systems are generally static and must support fine-grain parallelism. Subtle differences can make a large difference in the design of system software. For example, parallel systems can leverage the fact that the network is trusted, while distributed systems cannot.

Individual compute nodes should be as independent as possible. Nodes should be able to function on their own. A consequence of this is that the system software on one node should communicate with another node only when absolutely necessary.

Achieving high performance and scalability for applications requires that system software limits its use of resources as much as possible. In some cases, this means exposing low-level details of the system to the user. We attempt to mitigate this approach as much as possible using user-level libraries. It is often necessary to sacrifice transparency and portability to achieve the highest performance. This strategy also has the side effect of making the low-level system software less complex and easier to manage.

Finally, the simplest approach to system software is usually the best. Massively parallel computing is inherently complex. The system software should take steps to reduce complexity wherever possible. This ‘simple-is-better’ approach to system software has demonstrated high performance, has been shown to be highly scalable, and has been a key attribute in deploying and maintaining a reliable massively parallel computing system.

4.5. Software architecture implementation for Red Storm

We have slightly modified our software architecture implementation from the description given above specifically for Red Storm. Those changes are discussed below.

The Red Storm machine has single processor nodes, which eliminates the need for the multiprocessor modes that previous implementations of our lightweight kernels have provided. This functionality has been removed from Catamount. However, it is possible for future Red Storm systems to have up to four-way SMP nodes.

The Portals messaging architecture was designed specifically for the high-performance network interface on the Intel Paragon and ASCI/Red machines. For the Cougar implementation, Portals are data structures in user space that are traversed by the QK when a message arrives from the network. While we believe that this is an optimal implementation for those platforms, the lack of a functional programming interface for Portals limited its applicability to commodity high-performance networks or other networks where there is a programmable or intelligent network interface. For this reason, we developed a programming interface that allows Portals data structures to be placed in the optimal position for a given network [27]. For Red Storm, Portals structures exist partly on the network interface/router chip, where the processing of messages is performed. This allows the network interface to perform much like a message co-processor of the previous machines. A side effect of this change is that all Portals-related code can be removed from the QK, which further simplifies its implementation.

5. THE EARTH SIMULATOR

In order to make this paper relatively self-contained, a brief description of the Earth Simulator is included. This is taken from [3] essentially verbatim with only minor editing.

5.1. Hardware architecture

The Earth Simulator is a distributed-memory parallel computer system consisting of 640 processor nodes (PNs) connected by 640 $\times$ 640 single-stage crossbar switches. Each node is a system with a shared memory multiprocessor with eight vector-type arithmetic processors (APs), a 16 GB main memory unit (MMU), a remote access control unit (RCU), and an I/O processor. The peak performance of each AP is 8 Gflops. The Earth Simulator as a whole thus consists of 5120 APs with 10 TB of main memory and peak performance of about 41 Tflops.

Each AP consists of a four-way super-scalar unit (SU), a vector unit (VU), and main memory access control unit on a single LSI chip. The AP operates at a clock frequency of 500 MHz with some circuits operating at 1 GHz. Each SU is a super-scalar processor with 64 KB instruction caches, 64 KB data caches, and 128 general-purpose scalar registers. Branch prediction, data pre-fetching and out-of-order instruction execution are all employed. Each VU has 72 vector registers, each of which can have 256 vector elements, along with eight sets of six different types of vector pipelines: addition/shifting, multiplication, division, logical operations, masking, and load/store. The same type of vector pipelines works together by a single vector instruction and pipelines of different types can operate concurrently to the crossbar switches and receiving data. Thus the total bandwidth of inter-node network is about 8 TB s^{-1}. Several data-transfer modes, including access to three-dimensional (3D) sub-arrays and indirect access modes, are realized in hardware. In an operation that involves access to the data of a sub-array, the data are moved from one PN to another in a single hardware operation, and relatively little time is consumed in this processing.

The overall MMU is divided into 2048 banks and the sequence of bank numbers corresponds to increasing addresses of locations in memory. Therefore, the peak throughput is obtained by accessing contiguous data, which are assigned to locations in increasing order of memory address.

The fabrication and installation of the Earth Simulator at the Earth Simulator Center of the Japan Marine Science and Technology Center was completed by the end of February 2002.

5.2. Programming model

If we consider vector processing as a sort of parallel processing, then we need to consider three-level parallel programming to attain high levels of sustained performance for the ES.

The first level of parallel processing is vector processing in an individual AP; this is the most fundamental level of processing by the Earth Simulator. The compilers apply automatic vectorization techniques to programs written in conventional Fortran 90 and C.

The second level is that of shared-memory parallel processing within an individual PN. Microtasking and OpenMP are used to support a shared-memory parallel programming model. The microtasking capability is similar in style to that provided for Cray vector processors, and the same functionality is implemented for the Earth Simulator. Microtasking is applied in two ways; one (AMT) provides automatic parallelization by the compilers and the other (MMT) allows manual insertion of microtasking directives before target *do* loops.

The third level is distributed-memory parallel processing that is shared among the PNs. The distributed-memory parallel programming model is supported by MPI. The performance of this system for the `MPI_Put()` function of the MPI-2 specification was measured. The maximum throughput and latency for an MPI put operation is 11.63 GB s^{-1} and 6.63 μs, respectively. Only 3.3 μs is required for barrier synchronization; this is because the system includes a dedicated hardware system for global barrier synchronization among the PNs.

6. APPLICATION BENCHMARKS

In this section, we report on a series of single-processor benchmark studies comparing the SX-6 vector processor used in the Earth Simulator against a standard scalar microprocessor, a 2 GHz Intel Xeon. Although the Xeon is theoretically capable of two floating point operations per clock, in practice compiled code can only take advantage of one operation per clock in most cases. Therefore, we assume, for the purposes herein, one flop per clock and hence 2 Gflops peak, while the SX-6 processor has a theoretical peak of 8 Gflops [28]. Thus naively we might expect the SX-6 to outperform the Xeon by a factor of four.

In more detail, we note that the Xeon was configured with 512 KB of L1 cache and ran a Linux kernel (v2.4.18); applications were built using the Intel version 7.0 compilers. The SX-6 system used is located at the Arctic Research Supercomputer Center, consists of eight processors with a memory of 64 GB shared by all eight processors, and runs the Super-UX variant of Unix; NEC compilers were used. Just one of the eight processors was used in these studies.

6.1. Linpack

One benchmark frequently used to characterize the performance of processors is the Linpack Benchmark [19]. Linpack was compiled with generically available Basic Linear Algebra Subroutines (BLAS) and with vendor-supplied BLAS. Table V shows the data for both systems. From these data we can compute the speed of the SX-6 relative to that of the Xeon, as shown in Table VI.

The results in Table VII show that for the SX-6 to compete on a problem similar to Linpack the vector unit must be employed.

We note also that although Linpack is a problem that vectorizes well, the SX-6 is not dramatically outperforming the Xeon on percentage of peak computational rate for large vectors.

Table V. Linpack speed as a function of processor and BLAS type.

Processor	Solver type	BLAS source	BLAS type	Gflops ($n = 100$)	Gflops ($n = 1000$)
Xeon	Lapack	Vendor	Scalar	0.969	1.65
Xeon	Linpack	Compiled	Scalar	0.710	0.249
Xeon	Linpack	Vendor	Scalar	0.633	0.249
SX-6	Lapack	Vendor	Vector	1.16	7.76
SX-6	Linpack	Compiled	Vector	0.260	1.338
SX-6	Linpack	Vendor	Vector	0.309	1.338
SX-6	Linpack	Compiled	Scalar	0.092	0.108

Table VI. Linpack speed of the SX-6 relative to that of the Xeon as a function of BLAS type.

Solver, BLAS type	SX-6/Xeon ($n = 100$)	SX-6/Xeon ($n = 1000$)
Lapack, Compiled vector/scalar	1.198	4.7
Linpack, Compiled vector/scalar	0.37	5.37
Linpack, Vendor vector/scalar	0.49	5.44
Linpack, Compiled both scalar	0.14	0.43

Table VII. Approximate percentage of peak performance achieved on Linpack for the two processors in their best case.

Processor	% Peak ($n = 100$)	% Peak ($n = 1000$)
SX-6	14.5	95
Xeon	48.4	82.5

6.2. Livermore Fortran Kernel results

Another benchmark developed by McMahon of LLNL is the Livermore Fortran Kernels (LFKs), known colloquially as the ‘Livermore Loops’ [29]. This benchmark was assembled from a number of the kernels used in simulation codes in use at LLNL. The kernels are exercised for several different problem sizes from within a single code. Because the benchmark contains a number of kernels, no one characterization of net performance is appropriate to all cases. McMahon indicates a preference for the geometric mean of all kernels hence we have adopted that approach.

Table VIII. Speed on Livermore Loops in Gflops using geometric mean over all kernels.

Processor	LFK speed (Gflops)	Relative speed
SX-6 vector	0.407	1.12
Xeon	0.364	1.0
SX-6 scalar	0.113	0.31

The results in Table VIII show that for this metric the Xeon is quite competitive with the SX-6. However, there are some kernels on which the SX-6 excels, so for applications rich in these operations the geometric mean will greatly under-predict performance. Hence we examine a fragment of the kernel (23) for which the SX-6 performance advantage is the largest. We find this to be a vector evaluation of a Planckian distribution taken from a hydrodynamics code:

```
do
     w(k) = x(k)/(exp(Y(k)) - fw)
enddo
```

As the SX-6 provides hardware support for vector division while the Xeon does all division in software, the performance discrepancy is easily understood.

We next examine the kernel and span for which the SX-6 performance advantage is the smallest (0.07). Clearly, this cannot be vectorized:

```
do i=2,n
     W(i)=.01
     do k=1,i-1
          W(i) += B(i,k)*W(i-k)
     enddo
enddo
```

The single best performer was the following kernel taken from a hydrodynamics code:

```
do k
     x(k) = q+ y(k) * (r*Z(k+10)+t*Z(k+11))
enddo
```

On this, the Xeon obtained 59% of its peak performance, and the SX-6 was 1.4 times as fast. More generally, one observes that the mean expressed as a percentage of peak performance across all the LFK benchmarks is higher for the Xeon than for the SX-6, and one sees a much wider variation in performance for the SX-6 than for the Xeon.

Benchmarks are valuable, but do not tell the entire story. The full application must be run in order to obtain reliable performance information. In studying the performance of the SX-6, we chose to use the

Table IX. CTH grind time on various processors.

Machine	Grind time (μs)	Performance versus ASCI Red
ASCI Red	90.5	1.00
2 GHz Xeon	11.5	7.86
900 MHz Itanium II	12.8	7.07
SX-6 (vector)	10.1	8.96
SX-6 (scalar)	49.4	1.83

same applications and test problems that were used in evaluating candidate platforms for Red Storm. Of the four applications used, two have been ported and are working on the SX-6. We discuss results from these applications below.

6.3. CTH results

CTH [30] is a multi-material, large deformation, strong shock wave, solid mechanics code developed at Sandia National Laboratories. CTH is a finite difference code. It has models for multi-phase, elastic, visco-plastic, porous and explosive materials. Three-dimensional rectangular meshes; two-dimensional rectangular, and cylindrical meshes; and one-dimensional rectilinear, cylindrical, and spherical meshes are available. CTH uses second-order accurate numerical methods to reduce dispersion and dissipation and produce accurate, efficient results.

For the Red Storm evaluation, a test problem was created that met the following criteria: (1) it had to stress processor performance (minimal I/O); (2) it had to run on ASCI Red (which has 256 MB of memory per 2 CPU node). Based on these requirements, a test problem known as Cframe205 was generated. Cframe205 was also used to assess performance on the SX-6.

The traditional figure of merit used when measuring performance of CTH is the *grind time*, defined as the time to iterate a single cell of the mesh through one time step. Grind time is proportional to the reciprocal of the more familiar metric of scaled speedup. Table IX shows CTH performance data for several systems of interest.

The scalar result from the SX-6 is included to further underscore how the performance depends upon the use of the vector processor. Using the F_PROGINF utility on the SX-6, one can obtain more information about the utilization of the vector units. CTH utilized the vector units 20% of the time, and the average length of the vectors used was 61. Thus performance suffers from both the scalar portions of the calculations and the under-utilization of the vector units when they are being used.

Might one do better? Increasing the problem size would presumably increase the length of the arrays being passed to the vector units, and improve performance. Because CTH is a structured-grid code, this experiment is straightforward. The problem size was increased by a factor of four by doubling the number of cells in X and Y, and rerun on both platforms. Monitoring the memory during execution confirms that both jobs are now consuming over 700 MB. The grind times are now 9.4 μs for the SX-6 and 12.3 μs for the 2 GHz Xeon. The average vector length rises to 88, but the time spent in the vector

Table X. Profile of CTH routines on the SX-6.

% Time	Seconds	Cumulative seconds	No. of calls	μs per call	Name
19.6	14.9	14.9	900	16.56	eleb_
11.3	8.59	23.49	1800	4.769	elde3_
5.9	4.51	27.99	7 289 920	0.000	diatom_volume_fraction
5.8	4.43	32.42	900	4.92	dimin_
5.0	3.78	36.19	900	4.20	elygp_
4.1	3.14	39.33	880	3.57	erpy_
4.1	3.10	42.43	77 440	0.040	erfaxs_
3.9	2.96	45.40	929 280	0.0032	convcy_
3.8	2.88	48.28	900	3.21	eosmap_
3.4	2.59	50.87	79 200	0.0326	erfaz_

units remains at approximately 20%. For this problem we see the SX-6 outperforming the Xeon by approximately 30%.

Other experiences with the SX-6 indicate that substantial gains can be obtained by performance analysis and optimization; what are the prospects for CTH? Examining the profile for CTH on the SX-6, the results given in Table X are obtained.

Based on Amdahl's law, optimizing the leading 10 routines completely would lead to only a 1.44 times speedup, and would require the editing (and major removal) of approximately 7000 lines of source.

Achieving major speedup for CTH on the SX-6 through optimization is then a labor-intensive process with dubious likelihood of success. At best, one might take the 30% improvement over the Xeon mentioned above, and raise it to a factor of approximately 100% ($1.3 \times 1.44 = 1.8$).

6.4. ITS results

ITS (Integrated Tiger Series) is a Monte Carlo radiation transport computer code which has been used to model objects in a radiation environment, such as X-rays. It was originally written in the 1970s, and continues to be used and modified to meet new needs at Sandia. As with CTH, for the Red Storm procurement a test problem was generated. Table XI shows the execution times for this test problem on the systems under discussion.

We see that ITS behaves poorly on the SX6, and that the vector and scalar versions of the application have comparable performance. This is not surprising, as the basic structure of the application is scalar: each particle to be tracked is iterated separately to completion before starting the next. There is little opportunity for vectorization.

Table XI. ITS execution time on various processors.

Machine	Time (s)	Performance versus ASCI Red
ASCI Red	2389	1.00
2 GHz Xeon	281	8.5
900 MHz Itanium II	2238	1.07
SX-6 (vector)	3618	0.66
SX-5 (scalar)	3903	0.61

7. PERFORMANCE COMPARISONS AND PROJECTIONS

We will now develop a model for the performance of various supercomputers on a key application of interest to DOE. The CTH shock hydrodynamics code is one of the most heavily used codes at Sandia. To compare the relative performance of Red Storm and the Earth Simulator, we first developed an analytical model for the performance of CTH on a sample problem and calibrated this model against the existing supercomputer ASCI Red.

CTH is a shock physics code. The full code simulates fully compressible shock hydrodynamics of complex, multi-material, solid, liquid, and gaseous fluids and structures with realistic materials properties in an arbitrary 3D domain. It provides advanced methods for adaptive mesh refinement (AMR) and accurate interface tracking, as well as advanced methods to suppress Eulerian numerical diffusion [30]. However, the sample problem considered here consists specifically of two gases in a cubic region without AMR. The sample problem has an initial interface running through the cubic region at a 45° angle to all axes. The two gases are initially confined to opposite sides of the interface and are moving towards the interface at a constant velocity of $t = 0$. The behavior for $t > 0$ is that the gases collide, forming a shock wave, and mix near the interface.

Computationally, CTH solves a coupled set of 3D finite difference equations. Assuming for purposes of exposition in this paper that all dimensions are cubes of integers, the supercomputer is a P^3 array, each processor of which owns an N^3 mesh of cells. The processors repeatedly compute different aspects of the problem's physics and exchange boundary values during a single time step before doing a global synchronization to compute the next time step value. In the two-gas simulation, there are 25 cycles of computing and boundary exchange before doing the global synchronization.

The analytical model and machine parameters are described in Appendix A. To demonstrate the validity of the model, Figure 2 shows the correlation between the model and actual runtimes on ASCI Red. The figure assumes each processor owns a 20^3 mesh of cells regardless of the number of processors.

We then evaluated the model with parameters appropriate to Red Storm and two sets of parameters for the Earth Simulator. Since Red Storm has not been built yet, the parameters (in Appendix A) are our best estimates based on the design and on contractual requirements. For one set of Earth Simulator parameters, we used published parameters where feasible, and used the results of CTH benchmarks on a SX-6 (same processor as Earth Simulator) at the Arctic Region Supercomputer Center. The benchmarks indicated that the Earth Simulator should get about 5% of peak efficiency on CTH. To address the possibility that CTH could be tuned to make better use of vectors, we created a second set of Earth

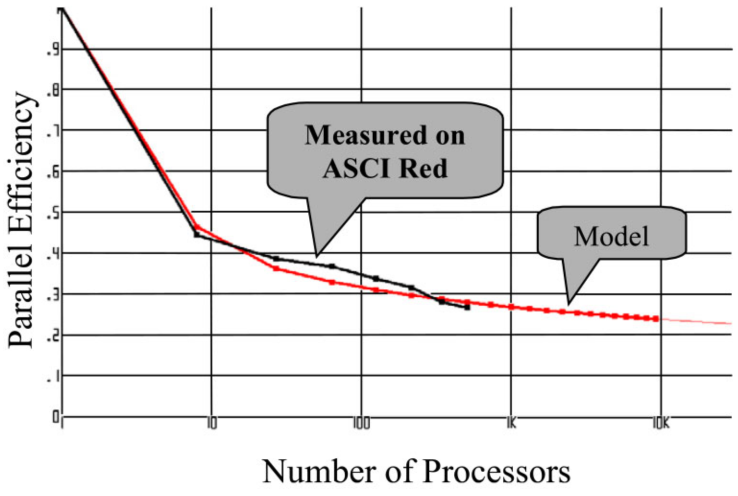


Figure 2. CTH CPU efficiency to calibrate the model to ASCI Red.

Simulator parameters with 15% of peak efficiency on CTH. The 15% peak efficiency is a best case scenario and has not been demonstrated. In fact, it exceeds our estimates (∼10% of peak) based on measurements reported in Section 6.3 of the best performance achievable for this application of CTH on the Earth Simulator's vector processors. Such a performance enhancement would require a major re-design of code data structures and algorithmic approach.

Figure 3 is a graph showing the overall performance of Red Storm and the Earth Simulator at 5% and 15% of peak efficiency. This graph is hypothetical in that it projects efficiency for all processor counts from one to over 30 000, even though the Earth Simulator only has 5144 processors.

The graph is not too surprising. The AMD Opteron used in Red Storm is somewhat newer than the SX-6 used in the Earth Simulator, and it stands to reason that it would outperform it on real benchmarks. However, if CTH could be vectorized, the vector architecture of the SX-6 would give it a boost over the Opteron.

Figure 4 adds the effect of cost by illustrating the number of useful CTH flops per dollar of machine cost. Figure 4 is based on cost parameters in Table AI in Appendix A. However, Figure 4 linearly scales the costs of the Earth Simulator and Red Storm by the number of processors. This is a simplification: it ignores the disproportionately large cost of increasing the Earth Simulator's crossbar as well as a quantity discount for purchasing a larger machine. In sum, it probably significantly under-estimates the Earth Simulator cost for very large machines.

The conclusions are striking: while the Earth Simulator and Red Storm are about equal in performance on a processor-to-processor basis, Red Storm costs quite a bit less per processor.

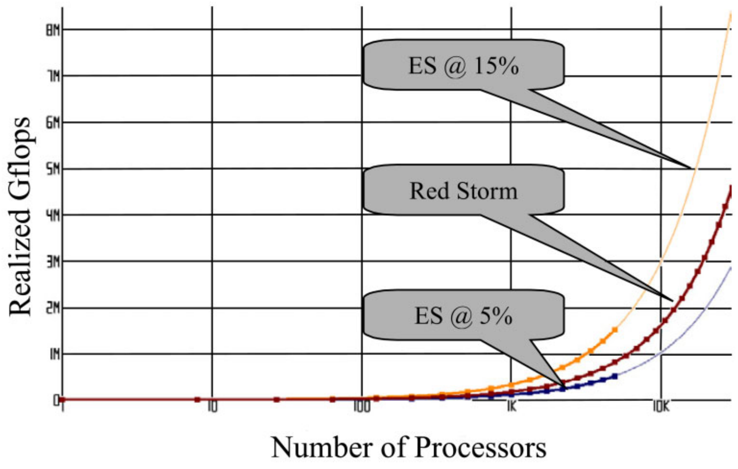


Figure 3. Overall system throughput by processor count.

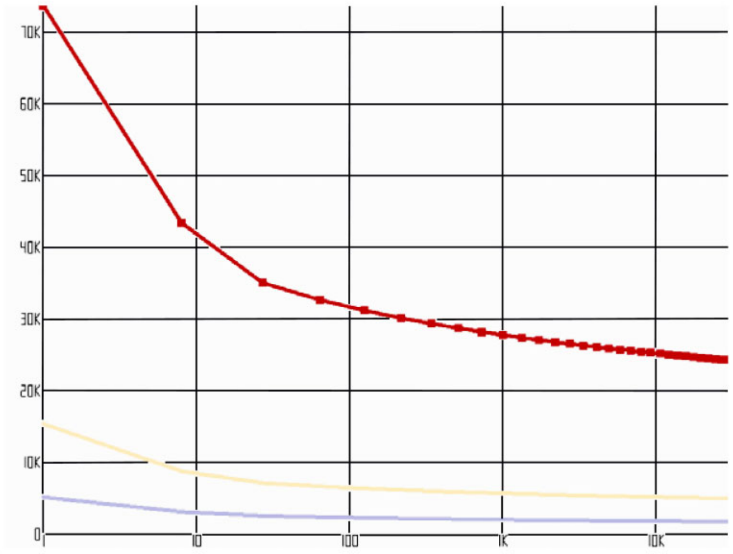


Figure 4. Realized flops per dollar.

8. CONCLUSIONS

In this paper, we have provided an overview of the Red Storm system architecture. This architecture has been developed based on our extensive experience designing and using large-scale, massively parallel systems on real-world science and engineering applications. We have also provided performance comparisons and projections that demonstrate the cost effectiveness of this approach compared with a large-scale vector-based machine, the Earth Simulator.

The Earth Simulator performs very well on a class of scientific applications that are well suited to vectorization. However, based on our testing of Sandia application codes on the NEC SX-6 vector processor and the Intel Pentium 4 scalar processor, we expect that Red Storm will outperform the Earth Simulator on these representative codes on a per-processor basis. We also expect that these codes will achieve higher parallel efficiency on Red Storm compared with the Earth Simulator.

In addition to single node and parallel performance, we expect that Red Storm will have a price/performance advantage over the Earth Simulator of a factor of two to three, after accounting for the differences in introduction dates for these systems. If the costs of space, power, and cooling are accounted for, the advantage may rise to a factor of four or more. The basic reason for the price/performance advantage of Red Storm is the use of high-volume commodity processors and a more cost-effective network topology.

We believe that the Earth Simulator represents a convergence of the traditional MPP design and vector design philosophies, and this convergence is a valuable development in the realm of advanced computing. We can also envision that other systems designed with this convergence could provide a common, commodity-like, infrastructure and supply vector nodes to some application communities and scalar nodes to others—or even allow mixing and matching scalar and vector processors in heterogeneous applications. Such an approach would serve to share the benefits and amortize the cost of supercomputers across a larger set of prospective customers and sponsors.

APPENDIX A

We assume for purposes of exposition that the total number of processors $P_{\text{total}} = P^3$, for an integer P. Each processor is assumed to 'own' an N^3 array of cells, for an integer N. The total problem size is therefore P^3N^3 cells.

The useful computational work in each time step is modeled analytically as C_0N^3, and the value of C_0 has been measured by 1 processor runs on ASCI Red as $C_0 = 3503$ flops. This yields

$$T_{\text{comp}} = C_0N^3/R_{\text{flops}}$$

However, it turns out that the work performed by each processor can go up or down depending on the physics within that processor. In other words, a processor can simulate two perfectly mixed gases at rest in less time than two gases in turbulent mixing. We model this by assuming C_0 is a random variable with a standard deviation C_1, and furthermore assuming that all the variances in C_0 are in phase during a time step in one processor. For the two-gas benchmark problem, we measure C_1 to be 1500. Thus, the additional time per time step due to this load imbalance will be

$$T_{\text{loadbalance}} = \Phi^{-1}(1 - 1/(P_{\text{total}} + 1))C_1N^3/R_{\text{flops}}$$

where Φ^{-1} is the inverse of the cumulative normal distribution.

Table AI. Model parameters.

Parameter	ASCI Red	Red Storm	Earth simulator 5% peak	Earth simulator 15% peak
R_{flops}	66 Mflops	640 Mflops	400 Mflops	1.2 Gflops
Lc	17 μs	3 μs	6.63 μs	6.63 μs
Bc	400 MB s^{-1}	6 GB s^{-1}	11.63 GB s^{-1}	11.63 GB s^{-1}
Cost @Procs	\$45 million @9460	\$90 million @10 368	\$400 million @5120	\$400 million @5120

CTH transmits all the data on each face as a single MPI message. The length of this message will be

$$S_{\text{msg}} = C^3(N+2)^2$$

where C^3 is 240 bytes for a two-gas problem.

The communications time will be

$$T_{\text{comm}} = 25\ N_{\text{maxfaces}} Lc + 25\ N_{\text{maxfaces}}\ S_{\text{msg}}/Bc$$

where N_{maxfaces} is the number of faces where communications occur for a mesh of P_{total} processors (e.g. for $N_{\text{maxfaces}} = 3$ when $P_{\text{total}} = 8$ because processors in a $2 \times 2 \times 2$ mesh only communicate on three of the six faces), and Lc and Bc are the MPI latency and bandwidth parameters.

CTH also performs a global synchronization each time step. It will be ignored here because its effect is on the order of 10^{-5}.

The total time per cycle is therefore given as $T_{\text{cycle}} = T_{\text{comp}} + T_{\text{loadbalance}} + T_{\text{comm}}$.

REFERENCES

1. Mattson TG, Scott D, Wheat SR. A TeraFLOP Supercomputer in 1996: The ASCI TFLOP system. *International Parallel Processing Symposium*, Honolulu, HI, 1996.
2. Tani K. *Earth Simulator Project in Japan* (*Lecture Notes in Computer Science*, vol. 1940). Springer: Berlin, 2000; 33.
3. Yokokawa M, Itakura KI, Uno A, Ishihara T, Kaneda Y. 16.4 TFLOPS direct numerical simulation of turbulence by Fourier spectral method on the Earth Simulator. *SC2002*, Baltimore, MD, 2002.
4. Sakagami H, Murai H, Seo Y, Yokokawa M. 14.9 TFLOPS three-dimensional fluid simulation for fusion science with HPF on the Earth Simulator. *SC2002*, Baltimore, MD, 2002.
5. Shingu S, Takahara H, Fuchigami H, Yamada M, Tsuda Y, Ohfuchi W, Sasaki Y, Kobayashi K, Hagiwara T, Habata S-I, Yokokawa M, Itoh H, Otsuka K. A 26.58 TFLOPS global atmospheric simulation with the spectral transform method on the Earth Simulator. *SC2002*, Baltimore, MD, 2002.
6. Boden N, Cohen D, Felderman RE, Kulawik AE, Seitz CL, Seizovic JN, Su W. Myrinet: A gigabit-per-second local-area network. *IEEE Micro* 1995; **15**:29–36.
7. Brightwell R, Fisk LA, Greenberg DS, Hudson TB, Levenhagen MJ, Maccabe AB. Massively parallel computing using commodity components. *Parallel Computing* 2000; **26**:243–266.
8. Navarro JP, Desai N, Evard R, Nurmi D. Scalable cluster administration—Chiba City approach and and lessons learned. *Proceedings of the IEEE International Conference on Cluster Computing*, Chicago, IL, 2002.
9. Brightwell R, Fisk LA. Scalable parallel application launch on Cplant. *SC2001*, Denver, CO, 2001.
10. Petrini F, Feng W-C, Hoisie A, Coll S, Frachtenberg E. The quadrics network: High-performance clustering technology. *IEEE Micro* 2002; **22**:46–57.
11. Partridge R. Cray launches X1 for extreme supercomputing, 2002. http://www.cray.com/downloads/crayx1_dhbrown.pdf.

12. MPI Forum, MPI: A message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing* 1994; **8**:165–414.
13. MPI Forum, MPI2: A message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing* 1998; **12**:1–299.
14. Maccabe AB, McCurley KS, Riesen R, Wheat SR. *SUNMOS for the Intel Paragon: A Brief User's Guide*. Intel Supercomputer Users' Group: Albuquerque, NM, 1994.
15. Shuler L, Jong C, Riesen R, van Dresser DW, Maccabe AB, Fisk LA, Stallcup TM. *The Puma Operating System for Massively Parallel Computers*. Intel Supercomputer Users' Group: Albuquerque, NM, 1995.
16. OpenMP Architectural Review Board. OpenMP Fortran Application Program Interface Version 1.0.
17. Amdahl GM. Validity of the single processor approach to achieving large scale computing capabilities. *American Federation of Information Processing Societies Spring Joint Computer Conference*, 1967.
18. Henning JL. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer* 2000; **33**:28–35.
19. Dongarra JJ. The LINPACK benchmark: An explanation. *First International Conference on Supercomputing*, 1988.
20. Standard Performance Evaluation Corporation. http://www.spec.org/cpu2000/.
21. HyperTransport Consortium. http://www.hypertransport.org/docs/spec/HT_IOLink_Spec.pdf.
22. Cray Research Inc. *SHMEM Technical Note for C, SG-2516*.
23. Gustafson JL, Montry GR, Benner RE. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing* 1988; **9**:609–638.
24. Greenberg DS, Brightwell R, Fisk LA, Maccabe AB, Riesen R. A system software architecture for high-end computing. *SC'97: High Performance Networking and Computing*, San Jose, CA, 1997.
25. Maccabe AB, Riesen R, v. Dresser DW. Dynamic processor modes in Puma. *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)* 1996; **8**:4–12.
26. Zajcew R, Roy P, Black D, Peak C, Guedes P, Kemp B, LoVerso J, Leibensperger M, Barnett M, Rabii F, Netterwala D. An OSF/1 UNIX for massively parallel multicomputers. *Winter USENIX Technical Conference*, 1993.
27. Brightwell R, Riesen R, Lawry B, Maccabe AB. Portals 3.0: Protocol building blocks for low overhead communication. *2002 Workshop on Communication Architecture for Clusters*, 2002.
28. Cray Inc. Cray SX-6 Product Overview, 2002.
29. McMahon F. The Livermore FORTRAN Kernels test of the numerical performance range. *Performance Evaluation of Supercomputers* (*Special Topics in Supercomputing*, vol. 4) Martin JL (ed.). North-Holland: Amsterdam, 1988; 143–186.
30. Hertel ES, Bell RL, Elrick MG, Farnsworth AV, Kerley GI, McGlaun JM, Petney SJV, Silling SA, Yarrington L. CTH: A software family for multi-dimensional shock physics analysis. *Proceedings of the 19th International Symposium on Shock Waves*, Marseille, France, 1993.